# SZAKDOLGOZAT

**Mező Attila**

**Debrecen**

**2008**

**Debreceni Egyetem**

**Informatikai Kar**

# Webes alkalmazásfejlesztés:
# Report Request Form

Témavezető:

Kuki Attila

Egyetemi adjunktus

Készítette:

Mező Attila

Programozó

matematikus

## TARTALOMJEGYZÉK

# BEVEZETÉS

Mindenki érzékelheti, mennyire dinamikusan és gyorsan fejlődik a világháló. Mára már kimondatlanul is az információ elsődleges forrásává vált. Egyre több honlap születik, az ezeket fejlesztő és ezzel foglalkozó cégek száma is exponenciálisan növekedett az utóbbi évtizedben. A weblapok tartalma is egyre inkább közelít a multimédiás tartalom felé, a dokumentumok és az alapvető információk kezdeti megosztása után manapság már képeket, hang- és videófileokat is találhatunk a honlapok tartalma között. De nem csak ilyen formában változott a www világa. Eleinte statikus honlapokat használtak, melyek lényege, hogy mindenki számára ugyanazt a tartalmat állítják elő, ugyanaz az információ jelenik meg akárki is látogatja meg a weblapot. A statikus honlap előnyei között kell említenünk, hogy egyszerű, emiatt könnyű elkészíteni, nem szükséges hozzá akkora webfejlesztői tapasztalat és a hibákat is könnyebben megtalálhatjuk és kijavíthatjuk. Természetesen egyszerűsége hordozza a hátrányait, hisz robosztusabb webes rendszert lehetetlen statikus dokumentumokból létrehozni. Ezzel szemben a dinamikus weblapok már különböző tartalmat állítanak elő a látogatótól, egyéb körülményektől és feltételektől függően. Egy ilyen honlap sokkal személyre szabottabb, általában a felhasználói interaktivitás lehetőségével is gyakrabban találkozhatunk. Legtöbbször a dinamikus weblapok mögött adatbázisok állnak, és ezekből állítják elő a megfelelő tartalmat. A dinamikus weblapok legtetején állnak a webes alkalmazások, melyek a legösszetettebb feladatokat látják el.

Egyetemi tanulmányaim vége fele a szakmai gyakorlat megszerzése céljából jelentkeztem egy debreceni érdekeltségű szoftverfejlesztő cég gyakornok-programjába. Céljuk az volt ezzel a programmal, hogy egyetemi tanulókat „tanítsanak be” az általuk használt technológiára és eszközökre. A tanulók is profitálnak, hisz amellett hogy pénzt keresnek, szakmai tapasztalatra tesznek szert és munkahelyet is biztosítanak maguknak már az egyetem elvégzése előtt. A cég pedig, amikor felveszi őket teljes munkaidőbe (a program ideje alatt a diákok részmunkaidőben dolgoznak), már ismerni fogja az új alkalmazottakat, és biztos, hogy a cég számára megfelelő szakmai háttérrel fognak rendelkezni. Így kerültem a ma már Roc Development Hungray Kft.-nek nevezett számítástechnikai céghez, mely egy nagyobb cégcsoport tagja, és központja az

Amerikai Egyesült Államokban van, Debrecenben a fejlesztési központ működik. Több webes alkalmazás elkészítése után a gyakornok csapat kapta a feladatot, hogy egységes kommunikációs csatornaként üzemelő webes alkalmazást fejlesszen le. Az alapprobléma az volt, hogy ha valakinek valamilyen kimutatásra volt szüksége, akkor valamilyen úton-módon felvette a kapcsolatot a Reporting Team-mel (mely a kimutatásokat készíti), és kérte az adott report elkészítését. Tehát nem volt egységes kommunikációs csatorna, és a report állapota sem volt automatikusan nyomonkövethető. Ezt a problémát célozta meg a Report Request Form nevű webes alkalmazás, mely jól meghatározott követelményeknek kellett hogy megfeleljen. Szakdolgozatomban törekedtem összeszedni a projekt elkészítéséhez szükséges elméleti ismereteket, technológiákat, eszközöket, és ezeket először általános ismertetés után többé-kevésbé a projekten keresztül bemutatni. Úgy gondolom, hogy az informatika nyelve az angol, így azokon a helyeken, ahol a fordítás erőltetett lett volna, esetleg nem lett volna túl szerencsés vagy egy informatikában már elterjedt szakszóról van szó, meghagytam a szó angol megfelelőjét.

# 1. ALKALMAZOTT ESZKÖZÖK, TECHNOLÓGIÁK

## 1.1 Webes alkalmazás

Ahogy a bevezetőben említettem, a statikus weblapok esetében klasszikus webes navigációról beszélhetünk a statikus dokumentumok között: linkek mutatnak az egyik oldalról a másikra, ezeket használva navigálhatunk a weblapon. Beszélhetünk még dinamikus weblapokról is, ahol a weplap tartalma (szöveg, kép, űrlapok mezői és egyéb megjelenő tartalmak) a körülményektől és feltételektől függően változhat. Ez a fajta dinamikus jelleg kétféleképp érhető el:

1. Kliens oldali script használatával megváltoztathatjuk egy adott lapon, hogy a felhasználói felület hogyan reagáljon az egér vagy a billentyűzet által kiváltott, illetve bizonyos időközönként kiváltódó események hatására. Ebben az esetben a dinamikus viselkedéssel a megjelenítésben, a felhasználói felületen keresztül találkozhatunk: tehát a meglévő tartalom megjelenítését, kinézetét változtatjuk meg. A leggyakrabban használt ilyen eszköz a Javascript. (Megj: természetesen a kliens oldali scriptek több lehetőséget is nyújtanak, de az egyszerűség kedvéért korlátozzuk alkalmazásukat csak ilyen feladatokra.)

2. Szerver oldali script esetén változhat az előállított és a böngészőnek elküldött oldal forrása (kérésről kérésre), hisz ilyenkor a weblap tartalmát alkotó elemeket határozzuk meg (mik és hol jelenjenek meg a lapon), amely bizonyos esetekben igencsak különböző lehet. A szerver által küldött választ (tehát a megjelenő weblapot) befolyásolhatják űrlapban küldött adatok, urlben kapott paraméterek, a használt böngésző típusa, az adatbázis vagy a webszerver pillanatnyi állapota is. Ilyen weblapokat többek között a

következő nyelvek segítségével állíthatunk elő: PHP, Perl, ASP, ASP.NET, JSP (a teljesség igénye nélkül).

A kliens oldali dinamikus tartalom természetesen a kliens gépén jön létre, a következőképp: a webszerver elküldi az adott lapot a böngészőnek („úgy ahogy van”), a böngésző végrehajtja a forrásba ágyazott kódot (legtöbbször Javascript) és megjeleníti a lapot a felhasználónak. Hátrányai közt annyit kell említenünk, hogy felhasználhatósága korlátolt, bár amire ki lett találva, arra jó is. A kliens oldali kód végrehajtása a böngészőt, és így a kliens gépét terheli, illetve elképzelhető, hogy például biztonsági okokból valaki letiltja a scriptek futtatását a böngészőjében, így nem érjük el a kívánt hatást és ez váratlan eredményekhez vezethet.

Szerver oldali dinamikus tartalomnál már másról van szó: a kliens böngészője küld egy http kérést a webszervernek. A szerver betölti a kívánt scriptet vagy programot, majd végrehajtja a kódot, mely általában egy html lapot állít elő. A program a dinamikus tartalomhoz szükséges információkat általában a query stringből vagy egy elküldött űrlapból kapja. Ezek után a szerver a kész html-t elküldi a kliens böngészőjének.

Ez a két technológia eredményezhet dinamikus weblapokat, bár legtöbbször a kliens oldali és a szerver oldali scriptek együttes, szimultán használatával találkozhatunk. Ezzel el is érkeztünk a webes alkalmazásokhoz, amik talán a legösszetettebb dinamikus weblapok közé tartoznak. Igazság szerint ezek olyan alkalmazások, amelyeket böngésző segítségével érünk el interneten vagy intraneten keresztül. Természetesen a megírásukhoz valamelyik böngésző által támogatott nyelvet használják (pl.: HTML, ASP, ASP.NET, PHP, JSP, stb.), és az alkalmazást maga a böngésző jeleníti meg. Közkedveltsége abban keresendő, hogy manapság már minden számítógépen található valamilyen böngésző (ezt gyakran vékony kliensnek hívják), mely a szerver-kliens kapcsolatban a kliens szerepét tölti be. Rengeteg érv szól a webalkalmazások használata mellett: nem szükséges külön kliens programot telepíteni, eltekintve az általános böngészőtől. Ha újabb verzió készül az alkalmazásból, a klienst nem kell frissíteni (csak a szerverre kell az új verziót telepíteni), a webes alkalmazás legközelebbi használatakor már a legújabb verzióval dolgozhatnak a kliensek is. A kliens gépén futó operációs rendszertől

függetlenül fejleszthetjük a webes alkalmazásokat (lehet az Mac OS, Windows, Linux). A funkcionalitás is egyre kevésbé korlátolt, a Java, a Javascript, a DHTML, a Flash és egyéb technológiák segítségével lehetőségünk van rajzolni a képernyőre, hangot lejátszani, illetve hozzáférünk a billentyűzethet és az egérhez is. A korábban említett kliens oldali eszközökkel pedig interaktívabbá tehetjük webes alkalmazásunkat, a felhasználó számára "láthatatlan" módon, azaz az oldal újratöltése nélkül frissíthetjük weblapunk tartalmát, ezzel is sokkal inkább közelítve egy asztali alkalmazás használatának érzetéhez, mely nagy arányban hozzájárul a felhasználó kényelméhez. Az alkalmazás "motorja" a webszerveren fut, így komolyabb számítások is elvégezhetők lassabb kliens gépeken is. A böngésző csak a végeredményt, és a számunkra szükséges információkat jeleníti meg, mi csak vezéreljük az alkalmazást a böngésző segítségével, a "lényegi munka" a szerver gépre hárul. A webes technológiák pedig – főleg a web térhódítása miatt – folyamatosan és szinte egyre gyorsabb ütemben fejlődnek, növelve a fejlesztők lehetőségeit és – nagyon gyakran – a felhasználók kényelmét (pl. AJAX). Természetesen minden éremnek két oldala van, így a webes alkalmazásoknak is vannak hátrányai. Talán az egyik legnyilvánvalóbb, hogyha nincs internet elérésünk, akkor az alkalmazást sem tudjuk használni. A böngészőnek is vannak korlátai, így minél inkább felhasználóbaráttá teszünk egy oldalt például Javascript segítségével, egyre többet kell a böngészőnek dolgoznia, ezzel pedig a kliens gépét terheljük. Általánossága miatt pedig nem helyettesíthet akármilyen asztali alkalmazást, nem minden erőforráshoz fér hozzá (elsősorban biztonsági okokból) és a vastag kliens kényelmes kezelését (például billentyűkombinációk) és gazdag felhasználói felületét is nehéz megközelíteni. Bár jóval kevesebb kompatibilitási probléma adódik egy webes alkalmazással kapcsolatban a böngészők között mint egy asztali alkalmazással kapcsolatban az operációs rendszerek között, azért nem szabad figyelmen kívül hagynunk ezt a problémát. Ez a HTML, a CSS és a DOM inkonzisztens implementációjának köszönhető, illetve a szabványok hiányának, pontatlanságának, félreértelmezésének és egyéb böngésző-specifikációknak. A felhasználónak is lehetősége van megváltoztatni a megjelenítési beállításokat (betűméret, szín, CSS és Javascript letiltása), ez pedig szintén a fejlesztői tervtől eltérő weblapot eredményezhet.

Mint láthatjuk, mind a vastag, mind a vékony kliensnek megvan a maga helye a szoftverfejlesztés terén, általában a követelmények segítségével dönthetjük el, épp melyik az, amelyik el tudja látni

a szükséges feladatokat. A Report Request Form esetén az intranet hálózat és az alkalmazás funkcionalitása miatt tökéletes választás volt a webalkalmazás. A fejlesztés ASP.NET technológiával készült (C# nyelven), a fejlesztőkörnyezet a Microsoft Visual Studio 2005 volt, az adatbázis szerver MS SQL 2000, a fejlesztés pedig a Scrum szoftverfejlesztési módszerrel történt.

## 1.2 ASP.NET

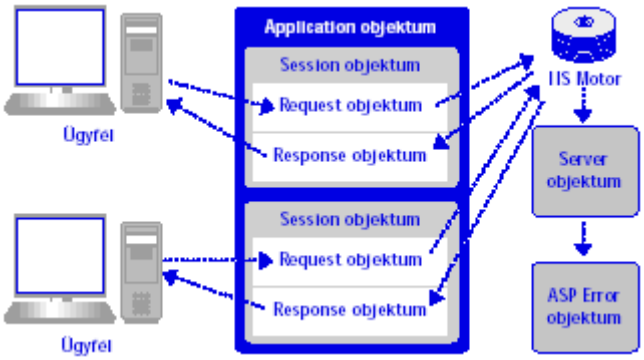
Mielőtt belekezdenék az ASP.NET leírásába, meg kell említenünk elődjét, a manapság már klasszikus ASP-nek nevezett technológiát, hogy lássuk, milyen irányban fejlődik a webes fejlesztés.

### 1.2.1 Klasszikus ASP

Az ASP a már korábban említett szerver oldali script nyelvek közé tartozik (bár inkább egy keretrendszer), tehát segítségével dinamikus weblapokat állíthatunk elő. Az alapötlet az volt, hogy milyen jó lenne, ha a HTML oldalak általában statikus részét hagyományos módon, pl egy WYSIWYG (What You See Is What You Got) szerkesztővel készíthetnénk, és csak a dinamikus részt kellene programozni. Az ASP ezt valósítja meg: statikus HTML elemek, script parancsok és COM komponensek használatával hozhatunk létre interaktív és könnyen lefejleszthető, módosítható webes alkalmazásokat. Ha a böngésző kér egy asp oldalt, akkor a webszerver feldolgozza lépésről-lépésre a kért asp filet, és ha talál benne script blokkot, akkor végrehajtja a benne található script parancsokat (szekvenciálisan). Az így előállított, a korábbi HTML részből és a script által esetlegesen visszaadott kódrészletből (amely szintén HTML kód) álló file-t küldi el a szerver a kliensnek, így a böngésző minden további nélkül fel tudja dolgozni (ha megfelelően készítettük el az asp oldalunkat, a script-rész sosem jut el a kliensig). Láthatjuk, illetve tudjuk, hogy ezen megvalósítás mögött egy interpreter áll, mely sorról-sorra értelmezi az asp fileban található script részleteket. Az ASP keretrendszerben a következő eszközök állnak

rendelkezésünkre: konstansok, változók, eljárások, függvények. Az .asp oldalak programozását, a HTTP kommunikáció, a webkiszolgáló egyes szolgáltatásainak elérését külön objektummodell segíti. Az ASP objektummodelljének minden elemét elérhetjük az .asp oldalak kódjaiból.

Az ASP objektummodell és annak elemei:




**1.ábra: Az ASP objektummodellje**

Ha a tranzakciós műveletekhez használt **ObjectContext** objektumot nem számítjuk, az objektummodell hat (Windows NT 4.0-n öt) objektumból áll. A Windows 2000-ben megjelent új objektum az **ASPError**, ami egy bekövetkezett hiba leírását tartalmazza, az .asp-be ágyazott, saját hibakezelést segíti.

A **Server** objektum magát az IIS-t képviseli (ASP.NET webszerver), néhány kiszolgálószintű beállítással és szolgáltatással. A Server objektum egy és oszthatatlan, és főleg kényelmi szolgáltatásai miatt hasznos. Pl.: Server.HTMLEncode(), Server.URLEncode(), Server.MapPath(), Server.Transfer(), Server.Execute(), Server.GetLastError(), Server.CreateObject().

Az **Application** objektum egy webalkalmazást jelképez. A webalkalmazás különálló egység, általában egy könyvtárban és annak alkönyvtáraiban található .asp kódok összessége, közös objektumokkal és beállításokkal. Az Application objektum segítségével a felhasználók, kérések között adatokat oszthatunk meg egyszerűen, anélkül, hogy például lemezre kellene írnunk azokat. Az Application objektumba írt információ mindaddig megmarad, amíg az adott alkalmazást (gyakorlatilag az IIS-t) újra nem indítjuk. Az ASP alkalmazás egy nagy globális objektum a felhasználók kérései és az azokra adott válaszok fölött.

A **Session** objektum egy ügyfél és a kiszolgáló között “fennálló“ kapcsolatot, munkamenetet jelképez. A “fennálló“ kapcsolatot azért írtuk így, mert valójában nem egy kapcsolatról van szó. Az IIS (a háttérben cookie-k segítségével) azonosítja a felhasználót és a böngészőt, így az a böngésző bezárásáig saját munkamenetébe térhet vissza. Az ASP munkamenet (Session) célja teljesen hasonló mint az Application objektumé, csakhogy az Application-nel ellentétben nem felhasználók közötti, hanem egy adott felhasználó műveletei fölötti globális objektum. Azaz, számos Request és Response fölött uralkodó objektum, ami megmarad egészen addig, amíg a felhasználó el nem hagyja a webhelyet, vagy be nem csukja a böngészőjét. Természetesen Session objektumból már nem csak egy van: ahány felhasználó, annyi Session. Az egyes munkamenetek azonosítása cookie-k segítségével történik. Az (egyébként változó) ASPSESSIONIDxxxxxxxx nevű cookie tartalmának segítségével az IIS egyértelműen azonosítja a visszatérő böngészőt, és ugyanabba a környezetbe helyezi (azaz, mindenki az első látogatáskor részére létrehozott, különbejáratú Session objektumba kerül).

Elbúcsúzhatunk a munkamenetektől, ha a felhasználó böngészője visszautasítja a cookie-kat. Ha ilyenkor mégis valami hasonló funkcionalitást szeretnénk elérni, külső gyártó termékeit kell használnunk, amelyek az URL-be ágyazva valósítják meg a munkamenetek kezelését (az IIS-ben szűrő- ként működve minden, az oldalakban található URL hivatkozáshoz hozzáfűzik az azonosítót, míg a böngészőktől érkező kérésekből kiszűrik azokat). Ez a megoldás teljesen cookie-mentes, és elvileg minden böngésző boldogul vele (csak kicsit rondák lesznek az URL-ek).

A Request objektum egy HTTP kérést jelképez, segítségével kódból hozzáférhetünk a kérés minden eleméhez, legyen az HTTP fejléc értéke, a böngészőben tárolt cookie, vagy kérdőív tartama. A klienstől kapott adatok pedig dinamizmus, az interaktivitás mozgatórugói. Adatok érkezhetnek URL-ben is (QueryString), ennek formája a következő:

**http://localhost/qs.asp?input1=val1&input2=val2**

Ekkor ilyen HTTP kérés indul a kiszolgáló felé:

**GET /request.asp?input1=value1&input2=value2&submit=Submit HTTP/1.1**

Ennek a módszernek több hátránya is van: egyrészt, a bemenő adatok növelik az URL hosszát, a kiszolgálónak elküldhető URL-ek mérete pedig biztonsági okokból általában korlátozva van. Másrészt nemcsak kényelmetlen, de nem is igazán biztonságos, hogy az átadott adatok

(amelyeket esetleg nem is mi írtunk be, hanem mondjuk egy kérdőív rejtett részei voltak) megjelennek a böngésző címsorában. Az átadott adatmezők név=adat formájúak (ld. fent: input1=val1), az egyes adatmezőket "&" jel választja el egymástól, az egészet pedig "?" a fájlnévtől (ez a sztring a QueryString). Egy adott mező értékét a **Request.QueryString("mezőnév")** függvény adja vissza. Ha az adatok között ilyen mező nem szerepel, a visszaadott érték (""").

Szerencsére a HTTP protokoll tartalmaz egy, a fentinél fejlettebb megoldást is. A POST parancs használata esetén a feltöltendő adatok a HTTP üzenet törzsrészébe kerülnek. Ha például egy FORM elem method attribútumát post-ra állítottuk, a következő kérés indul a kiszolgáló felé:

**POST /request.asp HTTP/1.1**
**Content-Type: application/x-www-form-urlencoded**
**Content-Length: 41**
**Connection: Keep-Alive**

**input1=value1&input2=value2&submit=Submit**

Láthatjuk, hogy a kiszolgálónak szánt adatok nem a POST parancs paraméterében, hanem a HTTP üzenet törzsrészében utaznak. A kérdőív (form) egyes mezőinek értékét a **Request.Form** kollekció tartalmazza.

A Response objektum pedig értelemszerűen a kérdésre küldendő választ jelképezi. Természetesen a Response objektum segítségével sem csak a válasz tartalmát, hanem a HTTP protokoll fejrészét is kezelhetjük.

Nézzünk pár dolgot a korábban már említett cookie-ról.

A cookie kis adatcsomag, amit a kiszolgáló kérésére a böngésző az ügyfél számítógépén tárol, és szükség esetén visszaküldi azt. Alapjában véve két különböző típusú cookie létezik: az első típus csak addig "él", amíg a böngészővel egy kiszolgálónál tartózkodunk. A böngésző bezárásával az ilyen cookie tartalma elveszik. Ezeket a cookie-kat elsősorban átmeneti célra használjuk (például az ASP Session fenntartására). A másik fajta, felhasználói szemmel gyakrabban megfigyelt cookie annyiban különbözik az előzőtől, hogy a böngésző bezárásakor nem veszik el, hanem a számítógép lemezére kerül. Az ilyen cookie-knak érvényességi idejük van. Amíg ez az

érvényességi idő le nem jár, addig a böngésző megőrzi az értéküket, és tartalmukat minden egyes látogatáskor visszaküldi a kiszolgálónak. A két cookie-fajtát csak a lejárati idő különbözteti meg egymástól.

Egy IIS kiszolgálón belül **Server** és **ASPError** objektumból egy-egy létezik (utóbbi csak akkor érhető el, ha hiba történt). **Application** objektum minden webalkalmazás egyedi, globális objektuma, **Session** objektum minden munkamenethez (ügyfélhez) egy jön létre, egyidejűleg tehát több is létezhet. **Request** és **Response** objektum pedig mindig az adott kérésre és válaszra vonatkozik, újabb kérés esetén új példány jön létre belőlük is.

### 1.2.2 ASP.NET

Az ASP.NET viszont egy igencsak más technológia. A célja ugyanaz, mint elődjének (és talán csak emiatt tekinthetjük elődjének a klasszikus asp-t), hogy dinamikus, interaktív weblapokat, webes alkalmazásokat hozzunk létre. Az ASP.NET mögött a .NET framework áll (innen is az elnevezés), mely alkalmazások fejlesztéséhez és futtatásához használt programozási modell, illetve szoftverfejlesztői platform. Két fő komponense van: a Common Language Runtime (CLR) és a Base Class Library (BCL). A CLR a .NET alapja, egy futtató környezet, a Common Language Infrastructure (CLI) Microsoft-féle implementációjának tekinthető. A CLI egy nyílt specifikáció, melyet a Microsoft fejlesztett ki, és leírja, hogyan kell kinéznie a futtatható kódnak és a futtató környezetnek. A specifikáció olyan környezetet határoz meg, melynek segítségével többféle magasszintű nyelvet használhatunk különböző számítógép platformon anélkül, hogy újra kellene írnunk bármit is a különböző architektúrák miatt. A CLI többek között a következő négy területre vonatkozóan tartalmaz leírást (illetve ezek az alrészei):

- Common Type System (CTS)
  Olyan típusok és a rajtuk végezhető műveletek halmaza, amelyen az összes CTS-nek megfelelő programozási nyelv osztozik. Meghatározza, hogy az adott típust (illetve annak értékét) hogyan kell tárolni a memóriában, így különböző nyelven írt programok könnyen megoszthatnak információkat. Célja, hogy olyan keretrendszer jöjjön létre, mely elősegíti a nyelvek közti integrációt és együttműködést, a típusbiztonságot és a gyors

programfuttatást. Tehát leírja, mi is pontosan és hogyan is néz ki egy osztály, egy struct, egy enum és bármi ilyesmi.

- Metadata

  Olyan Common Intermediate Language (CIL – erről madj később)-be ágyazott adatstruktúra, amely a kód magasszintű szerkezetét írja le. Minden olyan osztályról és osztálybeli tagról tartalmaz leírást, ami az adott assemblyben van definiálva, illetve amit az adott assembly egy másik assemblyből meghívhat. Egy eljáráshoz tartozó metadata tartalmazza a teljes leírását az eljárásnak, beleértve a tartalmazó osztályt és az azt tartalmazó assemblyt, a visszatérési típust és a paramétereket is. A .NET fordító legenerálja a metadata-t, és abban az assemblyben helyezi el, amely a köztes nyelvre (CIL) fordított kódot tartalmazza. Amikor a CLR elindítja a kódot, ellenőrzi, hogy a hívott metódushoz tartozó metadata ugyanaz-e, mint az, amit a hívó metódusban tárol. Ez biztosítja, hogy az adott eljárást vagy függvényt csak a megfelelő számú és típusú paraméterrel lehessen meghívni.

  Programozók is adhatnak metadata jellegű információkat a kódjaikhoz attribútumok segítségével. Az ilyen attribútumok olyan üzenetek a fordító számára, melyek alapján metadata-t állít elő.

- Common Language Specification (CLS)

  Olyan alapszabályok halmaza, amelyeket minden olyan nyelvnek be kell tartania, ami CLI-nek fordul, azért, hogy más, CLS-nek eleget tevő nyelvvel együttműködhessen. Ahhoz, hogy más objektummal is dolgozhassunk, függetlenül attól hogy milyen nyelvben implementálták őket, az objektumoknak csak azokat a tulajdonságaikat, jellemzőiket szabad “megmutatniuk” a hívó fél számára, amik közösek az összes olyan nyelvben, amivel kompatibilisnek kell lenniük. Pont emiatt hozták létre a CLS-t. Minden olyan komponens CLS-kompatibilis, amelyik eleget tesz a CLS szabályainak és csak a CLS-ben meghatározott jellemzőkkel rendelkezik. Ekkor garantált, hogy a komponens hozzáférhető, elérhető bármelyik CLS-t támogató nyelvben. A CLS-t úgy tervezték, hogy elég nagy legyen ahhoz, hogy az összes olyan nyelvi elemet tartalmazza, amire a fejlesztőknek szüksége lehet, de mégis elég kicsi ahhoz, hogy a legtöbb nyelv támogathassa. Továbbá minden olyan nyelvi elemet kihagytak a CLS-ből, amely

lehetetlenné tenné a kód típusbiztonságának gyors ellenőrzését, így lehetőség van arra, hogy minden CLS-nek megfelelő nyelvvel ellenőrizhető kódot állítsunk elő.

- Virtual Execution System (VES)
  Ez egy olyan környezet, mely felügyelt (managed) kód futtatását teszi lehetővé. Virtuális gépnek tekinthető, mely a CLI nyelvű program futtatásán kívül egyéb szolgáltatásokat is nyújt: memóriakezelés, szálkezelés, kivételkezelés, "szemétgyűjtés" (garbage collection), kódellenőrzés és kódbiztonság. Segítségével a fejlesztőknek nem kell foglalkoznia a CPU-specifikus dolgokkal, ezt a terhet leveszi vállukról az implementáció.

Nézzük hogy is működnek ezek a dolgok a gyakorlatban. A CLS tehát specifikációk halmaza, amely azt biztosítja, hogy ha az implementációban eleget teszünk a követelményeknek, akkor az így kapott nyelv és platform képes együtt dolgozni (garantáltan) más implementációkkal, tehát törekednek a mobilitásra, platformfüggetlenségre. Ezt valósítja meg Microsoft Windows© platformon a CLR, ami a neve ellenére nem „csak" a virtuális gép, ami futtatja a kódot, hanem az ehhez szükséges összes eszközt tartalmazza, legyen szó például a beépített osztályokról és típusokról, fordítóról, stb. Ahhoz hogy ez így működjön, szükség van egy köztes kódra, ez pedig a Common Intermediate Language (CIL). Az általunk valamilyen CLS-nyelven (C#, J#, Visual Basic, stb) megírt kód az előbb említett köztes nyelvre fordul, idáig hordozható a dolog, tehát natív kódról még nincs szó. Alapjában véve ez megfeleltethető a Javaban bevezetett bytecode-nak (bár ez inkább már gyűjtőfogalom – a CIL is bytecode), a CLR pedig többé-kevésbé a Java Virtual Machine-nek (JVM). A CIL (korábbi nevén MSIL – Microsoft Intermediate Language) platform- és processzorfüggetlen utasításokat tartalmaz, és a CTS illetve a metadata segítségével biztosítja az átjárhatóságot a nyelvek között. Amikor futtatni szeretnénk egy alkalmazást, akkor a köztes kód a CLR-hez kerül, amely átadja a Just-in-time (JIT) fordító alrésznek. A JIT fordítás futás idejű vagy dinamikus fordításként is ismert, célja a bájtkód-fordított rendszerek teljesítményének növelése. A JIT készít az eddigi CIL-ből natív kódot, amit az adott processzor már képes végrehajtani (a CLR minden támogatott CPU architektúra számára biztosít JIT fordítót). Ez a fordítás egyszer történik meg, futási időben, és ezután egy link jön létre a bytecode és a megfelelő fordított kód között. Hatékonysága abban rejlik, hogy nem szükséges az egész bytecode-t lefordítania natív kóddá, hanem ez alkalmazható fájl, függvény vagy tetszőleges

kódrészlet szintjén. Azt a tényt kell figyelembe venni, hogy bizonyos kódrészletek sosem futnak le egy program futtatása során, így lefordítani is felesleges ezeket (innen az elnevezés: Just-in-time – Épp-időben). Többek között ezzel próbálnak jobb teljesítményt elérni a JVM-hez képest. A forráskód feldolgozása, alapvető optimalizálása, fordítási időben történik. Tudnunk kell azt is, hogy a bájtkódról gépi kódra történő fordítás jóval gyorsabb, mint forráskódról. Természetesen a bytecode-ról natív kódra történő fordítás miatt szembesülnünk kell egy csekély késedelemmel a program első futtatásakor. Minél többet optimalizál a JIT, annál jobb kódot fog előállítani, bár ekkor a felhasználóknak többet kell várniuk az indításkor. Éppen ezért a JIT fordítónak kompromisszumot kell kötnie, hogy mennyi ideig tartson az optimalizálás és hogy milyen minőségű kódot állítson elő (pl. a JVM-nek két fő módja van – kliens és szerver. Kliens módban minimális fordítás és optimalizálás történik, hogy csökkentse a betöltési időt. Szerver módban viszont egy kiterjedt, átfogó fordítás és optimalizálás megy végbe, hogy maximalizálja a teljesítményt az alkalmazás futásához, ezzel feláldozva a betöltési időt). Létezik egyéb fordító is a CLR-ben a JIT-en kívül, ez pedig a Native Image Generator. Célja a sebesség növelése, mégpedig oly módon, hogy native image-eket hoz létre, melyben processzor-specifikus gépi kódra fordított állományok vannak, és ezeket az image-eket betölti a native image cache-be (ez a gépen található, a global assembly cache fenntartott része). Futtatáskor így nem kell JIT-tel lefordítani a köztes nyelvű programot, hanem használhatjuk a korábban létrehozott image-eket (sőt, ha egyszer létrehoztunk egy image-t, akkor futtatáskor eleve azt használja a CLR). Összegezve elmondhatjuk, hogy hátrányai és előnyei miatt mindkét fordítónak megvan a maga helye, körülményektől függően kell eldöntenünk hogy melyiket érdemes használni. Ejtsünk még pár szót a kód ellenőrzésről (code verification), mely a köztes nyelvről gépi kódra történő fordítás része. A CIL kódnak egy ellenőrzési folyamaton kell átesnie, amely megvizsgálja a kódot és a hozzá tartozó metadata-t, hogy megállapítsa, fennáll-e a típusbiztonság, azaz csak olyan memóriaterülethez férhet hozzá, amihez joga van. A típusbiztonság segít elkülöníteni az objektumokat egymástól, így védve őket a szándékos vagy nem szándékos károkozástól. Továbbá biztosítja egyéb biztonsági megkötések betartását is.

A .NET keretrendszer másik fontos alapja, a Base Class Library (BCL), mely egy olyan osztálykönyvtár, amely minden, .NET-et használó nyelv számára elérhető és osztályokat,

interface-eket és érték típusokat foglal magába. A programozók munkájának megkönnyítése érdekében a .NET keretrendszer a Base Class Library-ben rengeteg általános függvényt tartalmaz, mint például file olvasás és írás, grafikus megjelenítés, adatbázis kezelés és XML dokumentum módosítás. Kiterjedését tekintve sokkal nagyobb mint más nyelvek osztálykönyvtára, sőt, akár összehasonlítható a Java osztálykönyvtárával is. A BCL a keretrendszer minden verziójával frissül.

Ez az egységes rendszer a gyors és hatékony fejlesztést teszi lehetővé, legyen szó parancssoros, grafikus vagy webes alkalmazásról. Az újrafelhasználhatóság és egységesség miatt bármilyen korábban megírt komponenst felhasználhatunk (függetlenül attól, hogy melyik .NET nyelven írtuk), de a keretrendszer is igen gazdag osztálygyűjteményt tesz a fejlesztő alá, melyek segítségével minden általános programozási feladatra találunk megoldást. A keretrendszer fejlesztésével pedig egyre több osztályt használhatunk, melyek már speciálisabb problémákat céloznak meg.

### 1.2.3 Klasszikus ASP vs. ASP.NET

Láthatjuk, hogy az ASP.NET már egy új generációs webfejlesztési technológia a klasszikus ASP-hez képes. Vegyük sorba az alapvető különbségeket:

- A legfontosabb, hogy az ASP interpreteres nyelv, az ASP.NET pedig magasszintű nyelven írt, fordított osztályokra épülő technológia, így természetesen sokkal gyorsabb is
- Az ASP-nél minden kód „szem előtt” van, tehát nincs lehetőség a lényeges, esetlegesen „kényes” kódrészlet „elrejtésére”, míg az ASP.NET-nél erre ad lehetőséget a code-behind modell: az inline kóddal ellentétben, mely az ASP.NET oldalba közvetlenül beágyazott kódot jelent, a code-behind modell arra utal, hogy bizonyos kód az oldaltól (.aspx kiterjesztésű file) különálló fileban található (pl. c# nyelv esetén .cs kiterjesztésű). Ez szétválasztja az üzleti logikát a megjelenítéstől (erről majd később részletesen). ASP lapok esetében keveredve találunk script-részletet és HTML kódot a forrásban.

- ASP-nél a már korábban említett pár osztályt használhatjuk, míg az ASP.NET-nél több mint 2000 beépített osztály áll rendelkezésünkre.
- ASP-nek nincsenek szerver-alapú komponensei, míg az ASP.NET több szerver-alapú komponenst ajánl (Textbox, Button, stb) és az eseményvezérelt feldolgozás is történhet szerver oldalon.
- Az ASP nem támogatja az oldal szintű tranzakciót, míg az ASP.NET igen
- Az ASP.NET támogatja a mobil eszközökre történő webfejlesztést, mely lényege, hogy az eszköztől függően változik az előállított tartalom (wml vagy chtml, stb).
- Az ASP.NET teljesen objektum-orientált nyelveket használ és támogatja a nyelvek közti együttműködést
- Az ASP.NET támogatja a WebService-t

### 1.2.4 Code-behind modell

Az előbbi listában találkozhatunk a code-behind kifejezéssel. Egy ASP.NET weblap két részből áll: vizuális elemekből (HTML elemek, szerver oldali control-ok, statikus szöveg, stb) és az oldalhoz tartozó programozási logikából (eseménykezelés és egyéb kód). A single-file modell esetén ez a két rész egy file-ban található (innen is az elnevezés), egy .aspx kiterjesztésű file-ban (weblapok esetében). A kódrészlet egy script-blokkba kerül, melyben ugyanúgy használhatunk eseménykezelőket, metódusokat, property-ket vagy bármilyen más kódot, amit egy osztályt tartalmazó file-ban is használnánk. Ekkor ez az önálló file egy Page osztályból származó osztályként lesz kezelve, ez explicit nincs megadva, helyette a fordító hoz majd létre egy osztályt, melyben a felhasznált control-ok és eseménykezelők, metódusok, stb mint tagok (members) jelennek meg.
A code-behind modell esetében pedig lehetőség van a vizuális elemeket és a vezérlést végző kódot elkülöníteni egymástól két külön file-ba. Az egyik file tartalmazza a UI szintű elemeket, tehát többnyire HTML kódot, illetve ASP.NET control-okat (.aspx), a másik, mögöttes file pedig tartalmazza az adott .NET nyelven írt kódot: eseménykezelők, tagok, property-k, metódusok stb (.cs c# esetén). A kódfile egy teljes osztálydeklarációt tartalmaz, annyi kivétellel, hogy jelölve

van, hogy ez a file nem tartalmazza a teljes osztályt. Mindez hogy történik és mit is jelent, azt később láthatjuk a gyakorlatban.
A két megvalósítás között nincs különbség sebesség és funkcionalitás terén, így a választást más tényezőkre alapozva kell meghoznunk. Úgy gondolom, a single-file előnye csak annyi, hogy fizikailag egy file-ban tárolunk mindent (ennek összes, itt nem említett apró előnyével), így kisebb alkalmazások esetén áttekinthetőbb, könnyebben és gyorsabban módosíthatóbb. Code-behind esetében pedig nagyobb, összetettebb alkalmazásoknál különválik a megjelenítés a programozási logikától. Áttekinthetőbb lesz a forráskód, hisz nem egy 5000 soros, ömlesztett file-ban kell keresgélnünk ha módosítani szertnénk valamit, illetve egyszerre dolgozhat a designer a vizuális elemeket tartalmazó file-on, míg a programozó végezheti a lényegi kódolást a kódfile-ban. Ez a fajta szeparálás jelent meg az ASP.NET-ben az ASP-hez képest, így robosztusabb webalkalmazások is könnyebben fejleszthetők (természetesen ezt a mögötte álló, objektum-orientált keretrendszer is elősegíti). Ám még ennél is magasabb szintű absztrakciót valósít meg a 3 rétegű modell, mely az újabb alkalmazások közkedvelt architektúrája, így a Report Request Form esetén is alkalmazásra került.

## 1.3 A 3-rétegű modell

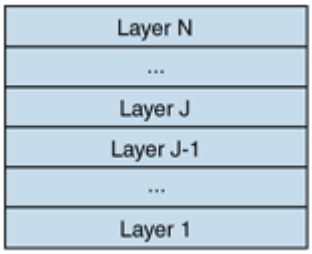
### 1.3.1 Általánosan, az N-rétegű modell

A probléma:
Előfordulhat, hogy olyan olyan összetett vállalati alkalmazást tervezünk, mely sok, különböző absztrakciós szintű komponensből áll. Problémát okozhat egy olyan struktúra létrehozása, mely támogatja a fenntarthatóságot, az újrafelhasználhatóságot, a skálázhatóságot, a robosztusságot és a biztonságot. Ezek érdekében törekednünk kell a következőkre:

- Valamelyik rész változtatása minimális hatással legyen más részekre, így kevesebb munka szükséges a hibafelderítéshez és –javításhoz, megkönnyíti az alkalmazás fenntartását és javítja az alkalmazás rugalmasságát

- Különítsük el egymástól a komponenseket (pl. a UI-t az üzleti logikától, az üzleti logikát az adatkezeléstől), mely növeli a rugalmasságot, a fenntarthatóságot és a skálázhatóságot
- A komponensek más alkalmazások által újrafelhasználhatóak legyenek
- Független csapatok dolgozzanak az egyes részeken
- Az egyenként önálló komponenseknek összetartozónak kell lenniük
- Az egymáshoz nem kapcsolódó komponensek legyenek lazán kapcsoltak
- A különböző komponensek egymástól függetlenül vannak telepítve, frissítve

A megoldás:
Válasszuk szét a komponenseket különböző rétegekbe. Ezekben a rétegekben a komponensek legyenek összefüggőek és legyenek nagyjából azonos absztrakciós szinten. Minden réteget az alatta található réteggel lazán kell kapcsolnunk. Kezdjük a legalacsonyabb absztrakciós szinttel, hívjuk ezt az 1. rétegnek. Ez a rendszer alapja. Haladjunk felfelé a létrán úgy, hogy a J. réteget helyezzük a J-1. réteg felé, egészen addig, míg el nem érjük a funkcionalitás legfelső szintjét – ezt hívjuk N. rétegnek.




**2. ábra: N réteg**

A többrétegű alkalmazás létrehozásának legfontosabb eleme a függőség kezelése. Egy adott szinten található komponensek csak azonos, vagy alacsonyabb szinten található komponensekkel léphetnek kapcsolatba. Ennek segítségével csükkenthetjük a különböző szinteken előforduló komponensek közötti függőséget. Két általános megközelítése van a rétegezésnek: szigorúan rétegzett (strictly layerd) vagy lazán rétegzett (relaxed layered).

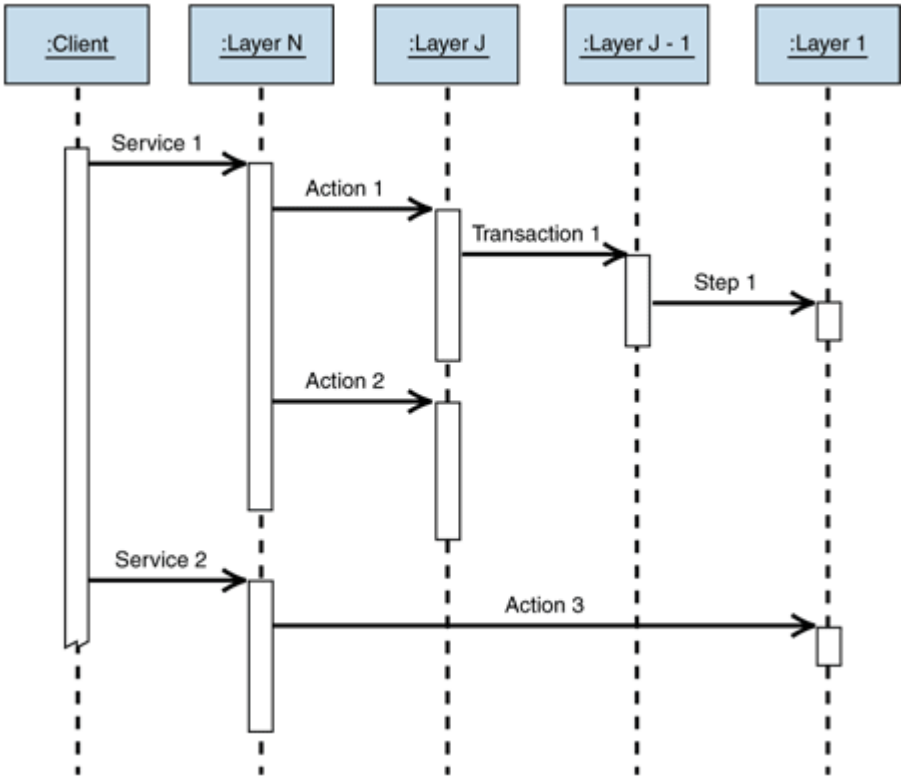
Első esetben egy komponens csak a vele azonos szinten lévő vagy a közvetlenül alatta található komponensekkel léphet kapcsolatba. Például az előző ábránál maradva a J. réteg a J-1. réteg komponenseivel, míg a J-1. réteg a J-2. réteg komponenseivel kommunikálhat, stb.

Egy lazán rétegzett alkalmazás annyiban lazít a megszorításokon, hogy egy komponens bármely alacsonyabb szinten található komponenssel kommunikálhat. Éppen ezért a J. réteg akár a J-1., a J-2. vagy a J-3. réteggel is kapcsolatba léphet. Ez egyrészt növelheti a hatékonyságot, mivel a rendszernek az egyszerű hívásokat nem kell az egyik rétegtől a következőig továbbítania, másrészt nem biztosítja ugyanazt az izolációs szintet, és így nehezebb egy alacsonyabb réteget úgy megváltoztatni, hogy az ne érintse a felsőbb rétegeket.

A többrétegű alkalmazásban alapvetően 2 módja van az interakciónak:

- Fentről lefelé
- Lentről felfelé

Fentről lefelé történő kommunikáció esetén egy külső elem a legfelső réteggel lép kapcsolatba. A legfelső réteg pedig az alsóbb rétegek szolgáltatásait használja, és ez így folytatódik, minden alacsonyabb szint a nála lentebb található réteget használja, egészen addig, míg a legalsó szintet el nem érik. Legyen a külső elem egy kliens alkalmazás, a többrétegű alkalmazás pedig egy szerver-alapú alkalmazás, mely szolgáltatásokat nyújt. A következő ábra szemlélteti a fentről lefelé történő kommunikációt:




**3. ábra: Kommunikáció N rétegű modellben fentről lefelé**

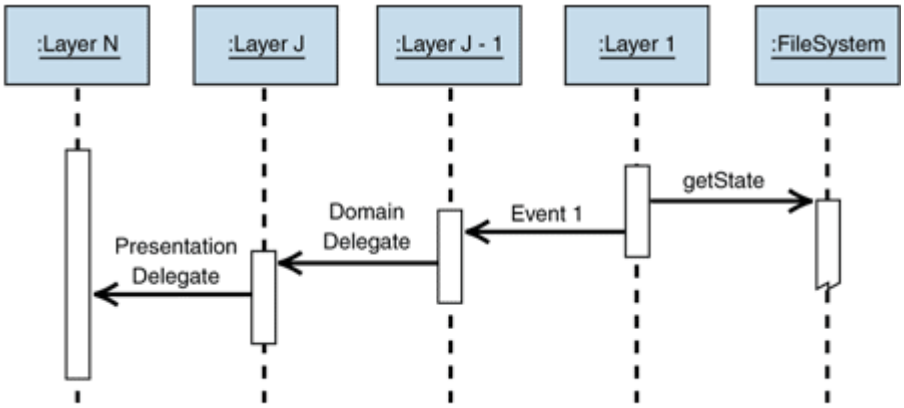
A kliens több szolgáltatást is használ, amit a szerver alapú alkalmazás nyújt. Ezeket a szolgáltatásokat a legfelső réteg adja, így a kliensnek csak ezzel a réteggel kell kommunikálnia és közvetlenül nincs tudomása az alacsonyabb rétegekről.

Lehetséges, hogy egyetlen bejövő hívás több kimenő hívást eredményez. Ez az eset látható a Service 1 meghívása esetén az N. rétegben. Ez gyakran előfordul, ha egy magasabb szintű szolgáltatás több alacsonyabb szintű szolgáltatás eredményeit gyűjti össze, vagy ha a magasabb szintű szolgáltatás vezérli több alacsonyabb szintű szolgáltatás bizonyos sorrendű végrehajtását.

Az ábra a lazán rétegzett megközelítést illusztrálja. A Service 2 végrehajtásakor átugorja az összes köztes réteget, és az 1. réteget hívja közvetlenül. Általános példa erre, ha a megjelenítésért felelős réteg (presentation layer) közvetlenül valósítja meg az adat hozzáférést, mellőzve

bármilyen, üzleti logikát megvalósító réteget (business logic layer). Adakezelő és –karbantartó alkalmazások gyakran ezt a megközelítést használják.
Egy legfelső szinten nyújtott szolgáltatás meghívása nem feltétlenül eredményezi az összes réteg meghívását. Ezt az elgondolást szemlélteti a Service 1 -> Action 2 folyamat, mellyel akkor találkozhatunk, ha egy magasabb szint önállóan fel tud dolgozni egy hívást, vagy ha rendelkezik még egy korábbi kérés eredményével.

Lentről felfelé történő kommunikáció esetén az 1. réteg észleli azokat az eseményeket, melyek érintik a felsőbb szinteket. Tegyük fel, hogy az 1. réteg az alkalmazást futtató szerver filerendszerének állapotát figyeli, ellenőrzi. A következő ábra egy tipikus lentről felfelé történő kommunikációt szemléltet:




**4. ábra: Kommunikáció N rétegű modellben lentről felfelé**

Az 1. réteg a helyi filerendszer állapotát vizsgálja. Ha változást észlel, akkor kivált egy eseményt, melyet a J-1. rétegben található egyik komponens kezel. Ez a komponens a J. réteg egyik callback delegate-jét hívja meg, melynek segítségével az adott réteg állapota frissül, majd erről értesíti az N. réteget, az N. réteg által erre a célra biztosított delegate segítségével. Mint ahogy az előző példánál, egy bemenet egy adott szinten itt is több kimenetet eredményezhet. Egy alacsonyabb réteg bármelyik magasabb réteget értesítheti, nem csak a közvetlenül felette lévőt. És végül, egy értesítés nem feltétlenül megy végig a teljes láncon.

Figyeljük meg, hogy a rétegek hogyan hatnak egymásra a lentről felfelé történő kommunikáció esetén, ellentétben a fentről lefelé történő kommunikációval. Az utóbbinál a magasabb szintű rétegek közvetlenül hívnak alacsonyabb szintű rétegeket, és éppen ezért függnek tőlük. Ha az interakció azonban lentről felfelé történik, akkor az alacsonyabb rétegek eventek, delegate-k és callback-ek segítségével kommunikálnak a magasabb rétegekkel. Erre az indirekcióra azért van szükség, hogy elkerüljük az alacsonyabb rétegek magasabb rétegektől való függőségét. Ha ez a függőség fennállna, akkor csökkenne a többrétegű alkalmazás által nyújtott előnyök száma.

Megvalósítás:

A megvalósításnak alapvetően két fajtája van:

- Létrehozzuk a saját rétegződési sémánkat
- Felhasználunk már egy létező rétegződési sémát

### 1.3.2 A 3 rétegű modell

A létező többrétegű alkalmazás-struktúrák közül az egyik általános, egyszerűsített séma a három rétegű alkalmazás modell, mely esetében a három réteg a következő:

- Megjelenítési réteg (presentation layer): általában a UI felületet jelenti, a felhasználó ezzel szembesül, célja a kommunikáció és az interakció lebonyolítása, illetve az adatok megjelenítése
- Üzleti réteg (business logic layer): ez a réteg valósítja meg az üzleti logikát, ez az alkalmazás „motorja”, közvetíti az adatot a felhasználói felület és az adatkezelő réteg között
- Adatkapcsolati réteg (data layer): ez a réteg tartalmazza azokat a kódokat, melyek biztosítják a különböző adattárolókhoz történő hozzáférést (pl. relációs adatbázishoz)

A 3-réteg között jól meghatározott kapcsolat áll fenn. Gyakran erre külföldi oldalakon 3-tier vagy 3-layer application-ként hivatkoznak, bár különbséget lehet tenni a tier és a layer szavak

használata között. Míg a tier inkább a fizikai elkülönítést jelenti, addig a layer esetében logikai rétegre bontásról beszélünk. Ez annyit jelent, hogy egy 3-tier alkalmazás esetében a különböző részek különböző számítógépeken, szerverken futnak, a 3-layer modellnél pedig az egész alkalmazás működhet egy gépen, itt a hangsúly a logikai csoportosításon van. A két kifejezés, mint már említettem, gyakran keveredik, így fontosnak tartottam a különbségek tisztázását.

Ez a 3 réteg bizonyos értelemben független egymástól, meghatározott módon kommunikálnak., mégpedig fentről lefele. Tehát a presentation layer csak a business layerrel lép kapcsolatba, a business layer pedig többnyire úgy szolgáltat vissza adatot, hogy csak a data layerhez intéz hívásokat, mely az adatokat szolgáltatja. Az adatkapcsolati réteg pedig a meglévő adatbázisból olvas vagy adatbázisba ír. Megj.: gyakran találkozhatunk más megközelítéssel is, mely szerint az alsó két réteg annyiban változik, hogy az általam említett adatkezelő komponens felkerül a business layer-be, és a data layert az maga adatbázis szerver alkotja.

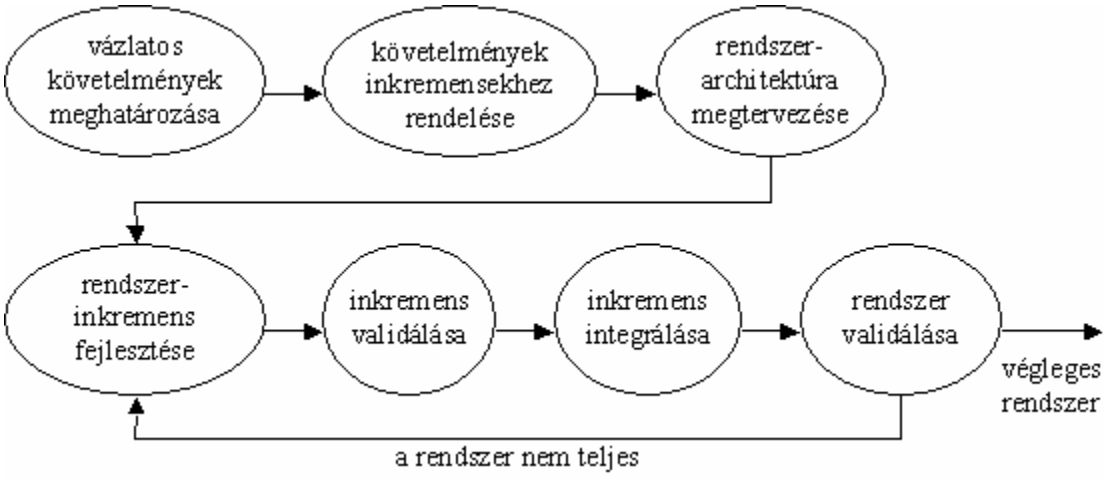
Ezek mellé épül fel egy megosztott könyvtár is (shared library), melyet mindhárom réteg ismer és használ, ezek többek között olyan osztályokat tartalmaznak, amelyek objektumai az úgynevezett business object-ek. A business object-ek képesek reprezentálni a számunkra fontos adatokat, és ezek „utaznak” a rétegek között, ezeken keresztül történik az információcsere. A business objectek nem mutatnak semmit az egyes rétegek működéséről, a megvalósítás ugyanúgy rejtve marad, a rétegek nem tudnak egymásról úgymond semmit. A modell azért jó, mert ugyanolyan alapokra építve, a rendszer magját meghagyva, alkalmazásunkat megírhatjuk más UI-ra is (pl. Windows Forms), ehhez csak a presentation layer-t kell lecserélnünk. Ugyanez a helyzet az adatbázissal is: ha megváltozik az adatbázisunk, akkor csak az adatmanipulációs műveleteket tartalmazó réteget kell újraírni, hogy megfeleljen az új adatbázisnak, a felsőbb rétegeket ez a változás nem érinti. Természetesen ezt az elméletet a tisztán szétválasztott rétegekről nehéz betartani, gyakran szembesülhetünk azzal, hogy a rétegek bizonyos ponton összemosódnak.

## 1.4 Scrum

A Scrum egy olyan iteratív, inkrementális szoftverfejlesztési modell, amely az agile szoftverfejlesztés egyik eszköze. Nézzük meg először általánosan, hogy mit is jelent az iteratív, inkrementális fejlesztési modell.

### 1.4.1 Iteratív, inkrementális fejlesztés

Mivel egy nagy szoftverrendszer fejlesztéséhez túl sok idő kell, így nem adható meg (tervezési) lépésben a teljes megoldás. Ráadásul a követelmények gyakran változnak a projekt fejlesztése alatt az adott szoftver és hardver megkötései és/vagy a befektető igényei miatt. Az iteratív fejlesztés lehetővé teszi, hogy folyamatosan finomítsuk a követelményeket és a nekik megfelelő szoftvermegoldásokat. Ideális esetben minden iteráció valamilyen „futtatható" eredménnyel zárul, így csökkentve a lehetséges kockázatok listáját, több teret ad a megbízói visszajelzéseknek, valamint a fejlesztők feladatai is jobban meghatározottak (kis iterációk).
A fejlesztés implementálás része kis méretű egységekben, inkremensekben történik, a mindenkori rendszerbe inkremenseket építünk, inkremensekkel bővítjük, és ha szükséges, nyilván megismételjük az inkremenshez akár a specifikációs tervezés lépését is.




**5. ábra: Inkrementális fejlesztés**

Az alsó rész, az inkremensek hozzáillesztése ciklikus folyamat, mindaddig folytatjuk, amíg úgy nem döntünk, hogy kész van a rendszer. Az inkrementális fejlesztésnek ez a ciklus a lényege. Az inkremenseket önállóan fejlesztjük, tervezzük, implementáljuk, validáljuk, sőt adott esetben még a követelményspecifikációt is megadhatjuk, megváltoztathatjuk inkremensenként. Az architektúrát változatlanul hagyva az inkremensek fejlesztése teljes életciklussal történik.

Az inkrementális modell előnyei:

- az inkremensek kicsik (úgy kell megtervezni, hogy kicsik legyenek), ezekre önmagukban a legmegfelelőbb modell alkalmazható, például vízesésmodell is (az inkremensek kicsik, jól körülhatárolhatók, adott esetben a követelmények egyértelműen specifikálhatók).
- az inkremensek fejlesztése történhet párhuzamosan
- kezdhetjük a fejlesztést a legfontosabb funkciókat realizáló inkremensekkel, és mivel az inkremensek kicsik, a felhasználó nagyon hamar kap egy nem eldobható prototípust, amely viszont már nem a követelményfeltáráshoz való, hanem már tudja használni. Tehát bizonyos funkciókat, a legalapvetőbb funkciókat tartalmazó rendszert nagyon hamar kap a felhasználó, a kevésbé lényeges funkciókat majd később beépítjük a rendszerbe. A rendszervalidálás mindig megtörténik, tehát a rendszer ilyen értelemben a beépített inkremensekkel önmagában egy részrendszer, tehát egy használható rendszer.
- a rendszerfejlesztés kockázata kisebb, mint az összes többi modellnél (a fejlesztések igen nagy része, több mint 50 százaléka sikertelen), mivel a rendszer validált, kis elemekkel, tehát van egy validált működő rendszer még akkor is, ha befullad a projekt, legfeljebb nem teljes funkcionalitással. Nagyon nagy a valószínűsége, hogy az első pár inkremensnél még nem fogy el a pénz, az idő, az ember, nem változik meg a környezet. Ezért a részsikeres befejezés valószínűbb ennél a fejleszési modellnél.
- a fontosabb funkciókat implementáljuk először, ennek a következtében azokat a funckciókat a felhasználók rendszeresen tesztelik, a fontosabb funkciókat jóval többször használják, mint a kevésbé fontosakat. Így már a rendszerfejlesztés közben a hibák hamarabb kiderülnek, a mindenkori részrendszerben valóban a legfontosabb funkciók a legteszteltebbek.

  Tehát egy robusztusabb rendszer fejlesztéséhez vezet ez a modell.

### 1.4.2 Az agile modell

Az agile folyamatmodell lényege, hogy a szoftver egy könnyebb, gyorsabb és sokkal ember-centrikusabb módon készüljön el. Vegyük sorba az agile módszer alapelveit:

- Megrendelő kielégítése használható szoftver gyors és folyamatos szállításával
- Működő alkalmazás gyakran áll elő (pár hetenként, mintsem pár havonta)
- A folyamat mérőegysége a működő szoftver
- Követelmények későbbi megváltoztatása is elfogadott
- Szoros, napi szintű együttműködés az üzleti emberek és a fejlesztők között
- Szemtől-szembeni beszélgetés a kommunikáció legjobb formája
- A projekt motivált egyénekre épül, akikben meg kell bízni
- Egyszerűség
- Önszervező csapatok
- Állandó alkalmazkodás a változó körülményekhez

### 1.4.3 A Scrum

Az agilis fejlesztés elveihez igazodik a Scrum metodológia is. A Scrum-ban különböző szerepkörök vannak. A szerepeket két csoportra oszthatjuk: disznókra és csirkékre.

Ez az elnevezés egy viccen alapul:

A disznó és a csirke mennek az utcán. Egyszer csak a csirke megszólal: „Te, nyissunk egy éttermet!” Mire a disznó: „Jó ötlet, mi legyen a neve?” A csirke erre gondolkozik, majd azt feleli: „Nevezzük *Sonkás tojás*nak!” A disznó erre: „Nem tetszik valahogy, mert én biztosan mindent beleadnék, te meg éppen csak hogy részt vennél benne.”

Tehát a disznók mindent beleadnak a szoftver megalkotásába, a csirkék pedig érdekeltek a projektben, de nem érdekli őket annyira, ha az elbukik. A csirkék igényeit, vágyait és ötleteit is figyelembe veszik, de nem oly módon, hogy azok akadályozzák a projektet.

Disznók:

A disznók tehát elkötelezettek a projekttel kapcsolatban, ők viszik vásárra a bőrüket (sonkájukat).

- Terméktulajdonos (Product owner): egy személy, aki a megrendelőt személyesíti meg, döntéseket hoz üzleti szempontból és a követelményeket illetően. Szinte mindig a csapat rendelkezésére kell állnia.
- Scrum Master: nem a csapat vezetője (mivel az önszervező), hanem az, aki segíti a folyamatot és segít elhárítani az akadályokat. Betartattja a szabályokat és felügyeli, hogy a fejlesztés úgy történjen ahogy tervezték. Célja a produktivitás növelése, bármilyen lehetséges módon. Feladata közé tartozik annak biztosítása is, hogy a csapat el legyen szigetelve a terméktulajdonostól.
- Csapat (Team): a gyakorlat azt mutatja, a legoptimálisabb a Scrum fejlesztéshez az 5-9 fős csapat. A csapatnak kell biztosítani, hogy a termék időben elkészüljön. Különböző feladatokat betöltő tagokból áll: fejlesztő, desginer, tervezők, teszterek, stb. A csapat dönti el, hogy mi a legjobb módszer a kitűzött cél elérésére, de övék a felelősség is. Gyakran egy helyiségben is vannak, ezzel is javítva a hatékonyságot és az emberközpontú fejlesztést.

Csirkék:

A csirkék nem részei az aktuális Scrum folyamatnak, de figyelembe kell venni őket. Az agilis szoftverfejlesztés egyik lényeges eleme, hogy a felhasználókat és a többi érintett embert is bevonják a folyamatba, építenek a tőlük érkező visszajelzésekre.

- Felhasználók (Users): azok, akik használni fogják a végleges terméket.
- Stakeholder-ek: azok az emberek, akik kellenek a szoftver létrejöttéhez, de nem vesznek részt közvetlenül a folyamatban.
- Menedzserek (Managers): olyan emberek, akik felállítják a körülményeket a szoftverfejlesztő szervezet számára.

Minden projektnél szükség van egy kezdeti tervezési fázisra. Ebben a fázisban kell egy architektúrát létrehozni, és ki kell nevezni egy fő tervezőt. A fejlesztés közben ez az architektúra változtatható (sőt, erre készen is kell állni), de kezdetben mindenképp szükség van egy tervre. A kezdeti tervezés után pedig rövid fejlesztési fázisok sorozata következik (egy ilyen fázis a *sprint*), melyek során inkrementálisan jön létre a termék. Egy sprint általában 1-4 hétig tart. A fejlesztés végén pedig van egy záró fázis. A csapat nyomon követi az összes rá váró feladatot (*task*), ezt egy listában kapják meg, ezt a listát hívják *backlog*-nak. Ez a lista irányítja a csapat munkáját. Két fajtája van, az egyik a *product backlog* és a *sprint backlog*. A product backlog tartalmazza az összes követelményt priorizálva. Funkcionális és nem-funkcionális követelmények is szerepelnek rajta, illetve csapat által létrehozott technikai követelményeket is tartalmaz. Bár láthatóan több helyről is érkeznek adatok, egyedül a terméktulajdonos felelőssége a prioritások meghatározása. A lista elemei a *product backlog item*-ek. Ezek olyan (munka)egységek, melyek elég kicsik ahhoz, hogy egy csapat egy sprint alatt befejezze. Ezeket az elemeket egy vagy több feladatra bontják. A sprint backlog pedig tulajdonképpen a product backlog egy része, azon feladatok halmaza, amelyeket a sprint során teljesíteni kell.

Minden sprint előtt, a csapat frissíti a backlog-ot, és (újra)priorizálja a benne lévő feladatokat. A csapattagok kiválasztanak egy számukra megfelelő feladatot, amit saját bevallásuk szerint képesek megcsinálni, majd becslést adnak a szükséges időt illetően. (Itt érdekességképp jegyezném meg, hogy a becslésnél használt idő feltételezi a hasznosan töltött órák számát, tehát azt az időt, amit konkrétan a feladat megoldására fordítunk. Így egy munkanap a mi fejlesztői csapatunkban 8 óra helyett 5 óra effektív munkaidőt jelent.) A becsléseket figyelembe véve, a ScrumMaster meghatározza a sprint végét. Ezt az időpontot nem lehet később módosítani, csak az adott sprintre tervezett funkcionalitáson lehet karcsúsítani. Ezt a folyamatot nevezik *sprint planning meeting*-nek. A fejlesztők naponta frissítik a kiválasztott feladat(ok) megvalósításából hátralévő órák számát, így frissítve a *sprint burndown chart*-ot. Ez egy grafikai megjelenítése a napra vonatkozóan a hátralévő órák számának. Ideális esetben a sprint végére a hátralévő órák számának értéke eléri a 0-t, és szépen egyenletesen csökken. A valóságban ez nem így van, többnyire a gráf fel-le „ugrál”, mivel a fejlesztés során új feladat is megjelenhet, és előfordulhat az is, hogy a kitűzött feladatokat nem lehet a sprint végére maradéktalanul teljesíteni.


**6. ábra: Sprint burndown chart**

A sprint során a csapat napi, úgynevezett *Scrum meeting*-eket tart, mindig meghatározott időben (a csapat dönti el, mikor). Ha valaki késik, vagy nincs jelen, akkor korábban a csapat által megbeszélt szankció kerül alkalmazásra (lehet akár csapatkasszába fizetett kisebb összeg, fekvőtámasz vagy akár gumicsirke viselése egész nap, szóval bármi, amiről korábban megegyeztek a csapattagok). Általában 15 percre tervezettek, a csapat méretétől függetlenül és többnyire állva tartják. Bárki részt vehet, de csak disznó szerepkörűek beszélhetnek. Többek között csapatépítő szerepe is van.

A megbeszélés során, mindenkinek 3 kérdésre kell válaszolnia:

- Mit csináltál tegnap óta?
- Mi a terved mára?
- Van valami akadály, ami meggátol abban, hogy megvalósítsd a célod?

Látható, hogy ezekre könnyen lehet válaszolni, mégis átfogó képet adnak a sprint és azon belül is a konkrét feladatok állapotát illetően. Fontos kiemeleni, hogy egy adott probléma vagy feladat konkrét megoldásáról nincs szó, nincs ún. *brainstorming*, a scrummaster feladata, hogy ezek a megbeszélések rövid ideig tartsanak, és fókuszáljanak a kitűzött kérdések megválaszolására.

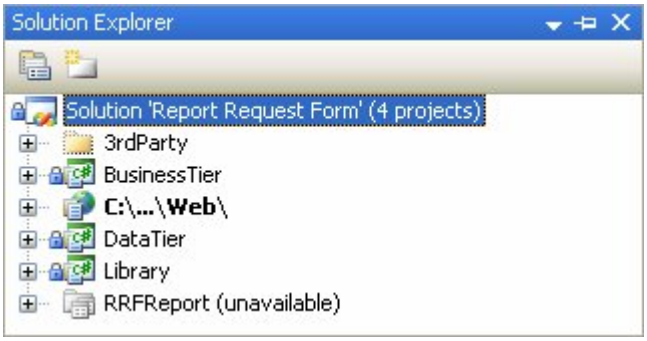
A sprint végén van a *sprint retrospective meeting*, mely célja, hogy a csapat és a scrummaster megbeszéljék, mi volt az, ami jól működött a sprint során, és mi az, amin javítani kell a következő sprintnél. Ezzel is javítják a hatékonyságot, növelik a csapat együttes teljesítményét.

A sprint lezárásakor a csapat szállítja az új verziót, tisztázzák mit sikerül megvalósítani és mi az ami kimaradt. Ekkor dönthetnek arról, érdemes-e folytatni a projektet, figyelembe véve az eddig megvalósított követelményeket és az aktuális körülményeket. Ha folytatódik a munka, frissítik a product backlog-ot, esetlegesen újrapriorizálják, és kezdődhet az új sprint. Ez így megy egészen a végleges termék elkészültéig (illetve annak határidejéig).

# 2. AZ ALKALMAZÁS

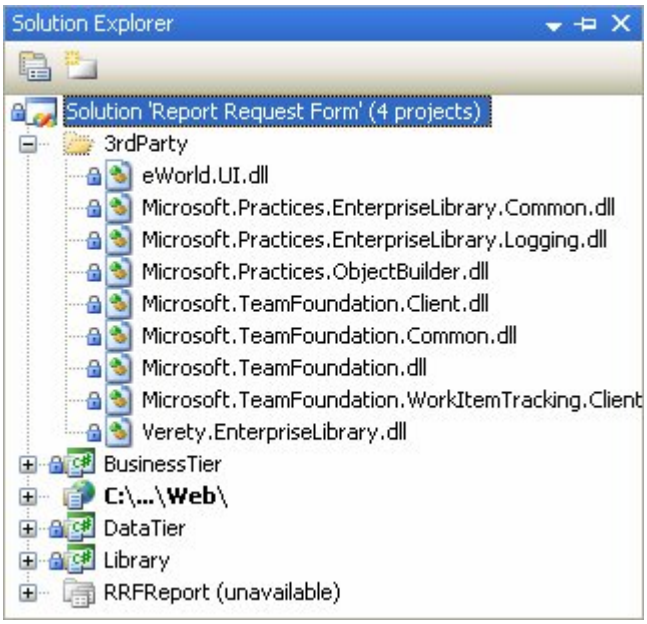
Az alkalmazás tehát Scrum metodológiával készült, 3 rétegű architektúrával, ASP.NET technológiával, .NET platformon és C# nyelven, Visual Studio 2005 fejlesztőkörnyezettel és MS SQL 2000 adatbázis szerverre.

## 2.1 Az alkalmazás felépítése




**7. ábra: Az alkalmazás felépítése**

Az alkalmazás 4 projektből áll, 3 osztálykönyvtárból  és 1 weboldalból. Van egy segédkönyvtár is, 3rdParty néven, melyben third-party eszközök találhatóak.




**8. ábra: 3rdParty komponensek**

Ezek más gyártó által lefejlesztett komponensek. Az eWorld.UI.dll tartalmaz egy naptárat, ami szinte minden fontos funkcionalitással rendelkezik. A többi komponens egyéb hasznos funkciókat valósít meg, mint például a logolás és a központi projektszerver elemeinek módosítása.

### 2.1.1 Shared Library

Láthatjuk, hogyan felel meg (többé-kevésbé) a többrétegű architektúrának ez a webes alkalmazás. Ennek szemléltetésére a User osztályokat használom. A Library az a megosztott osztálykönyvtár, mely az alkalmazásban előforduló, adat és üzleti szinten is megjelenő objektumok interface-eit tartalmazza. Ezekben vannak definiálva az alapvető tulajdonságok.

IUser: a rendszerben regisztrált, meghatározott szerepkörrel rendelkező felhasználó.

IRole: a felhasználó szerepköre a rendszerben.

IReportRequest: a kérést reprezentáló objektumot definiáló interface.

IRequestState: a report aktuális állapota.

IAttachment: olyan csatolt állomány, amely szükséges lehet a report megfelelő elkészítéséhez.

IReportingFrequency: milyen időközönként kell az adott report-ot előállítani.

IDistributionType: az előállított report hogyan legyen publikálva.

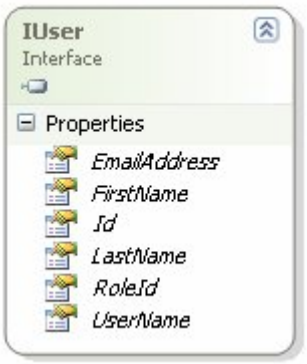
IDistributionFormat: ha a DistributionType értéke email, akkor ez határozza meg, hogy az emailben elküldött állomány (amely a report-ot tartalmazza), milyen típusú legyen.

IHeaderFooterElement: a report dokumentumban fejlécben vagy lábjegyzetben megjelenő információ.

IReportField: a reportban megjelenő egy mező.

IFieldValueType: a report egy adott mezőjéhez tartozó értékek típusa.

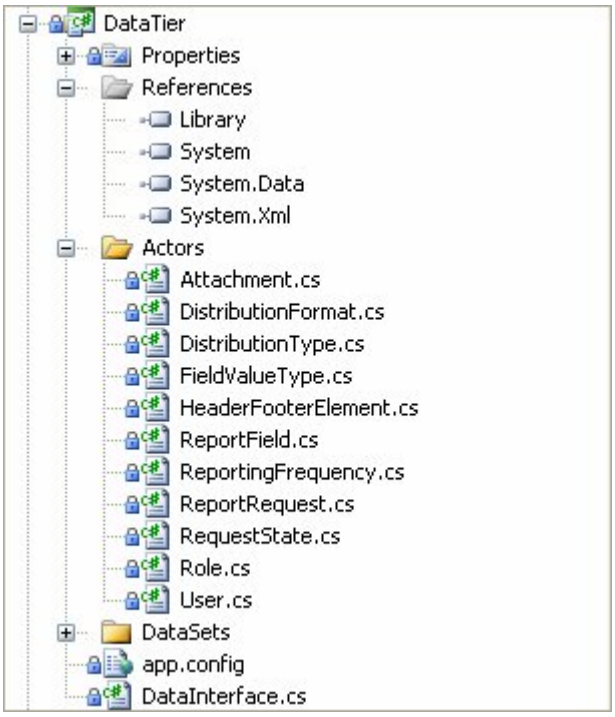
Példa: Library.Items.IUser.cs




**9. ábra: IUser interface**

### 2.1.2 Data layer

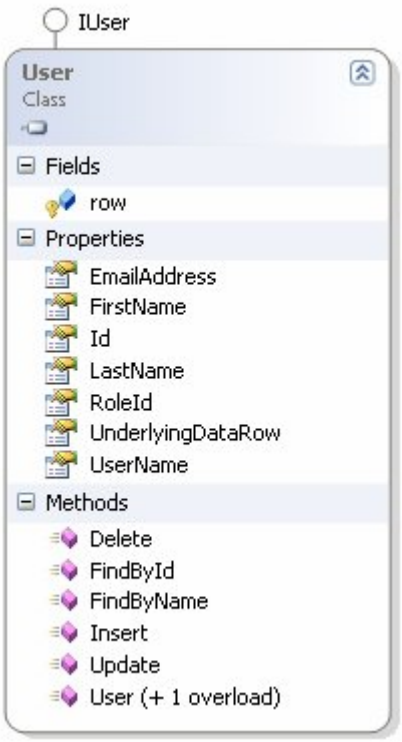
A Library osztálykönyvtárat referenciaként tartalmazza a DataTier osztálykönyvtár, mely a több rétegű architektúra adatkapcsolati rétegét valósítja meg.




**10. ábra: Az adatkapcsolati réteg**

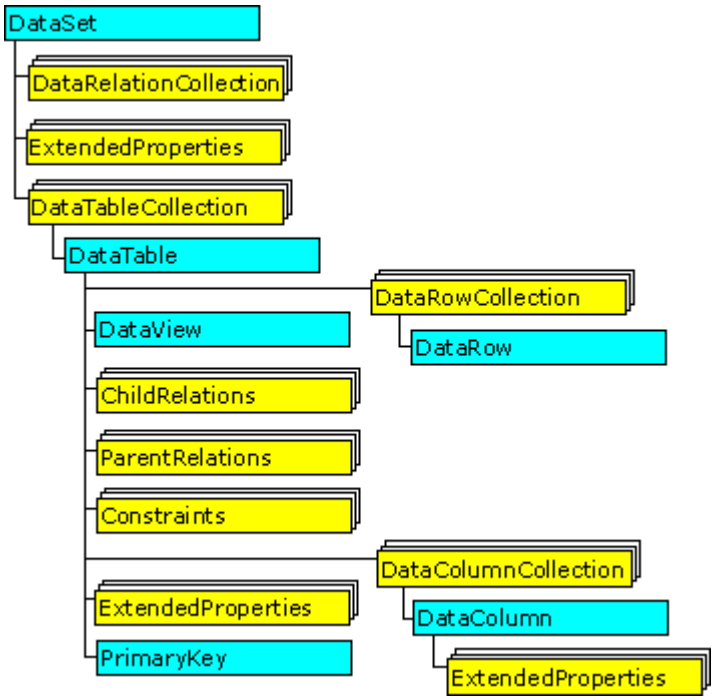
Az Actor könyvtárban az interface-ket implementáló osztályok vannak. Ezek hordozzák az adatbázisból kinyert adatokat.

Példa: DataTier.Actors.User.cs




**11. ábra: DataTier.Actors.User.cs osztály**

A DataInterface.cs tartalmazza azokat a függvényeket, melyek Dataseteken keresztül hívnak adatbázisbeli tárolt eljárásokat. De ebben az osztályban csak azok a függvények jelennek meg, amik gyűjtemény típusú visszatérési értékkel rendelkeznek, tehát egy adatbázisbeli tábla több sorát ad(hat)ják vissza. Azokat a tárolt eljárásokat, amelyek egy konkrét sort választanak le az adatbázisból, a megfelelő actor osztályán keresztül hívhatjuk meg. Így ha például adott szerepkörű felhasználókat keresünk, akkor a DataInterface `SelectUsersByRoleId(int roleId)` metódusát kell meghívni, ám egy konkrét felhasználót a User osztály `FindByName(string userName)` függvényével kereshetünk ki. Mindkét függvény típusos Dataset-tel dolgozik. A Dataset osztályok olyan hierarchikus csoportja, mely memóriában tárolt relációs adatbázist ír le. Ezek az osztályok reprezentálják az adatbázisbeli táblákat, a köztük lévő kapcsolatot, a megszorításokat, a táblák sorait és oszlopait, stb.




**12. ábra: Dataset osztályhierarchiája**

Az ADO.NET (a Base Class Library azon része, mely az adatkezeléshez szükséges osztályokat tartalmazza) data provider-en (közös interface) keresztül biztosít konzisztens hozzáférést a gyakori adatforrásokhoz. Minden támogatott adatforráshoz léteznek külön provider objektumok, melyek az adott adatforráshoz vannak tervezve. A data provider osztályai többek között:

- Connection: az adatforrással történő kommunikációért felelős.
- Command: az adatbázison hajthatunk végre különböző műveleteket.
- Parameter: egy parancs egy paraméterét írja le.
- DataAdapter: az adatforrás és a Dataset közti adatszállításért felelős.
- DataReader: adatbázisból nyert rekordok feldolgozását támogatja.

Ezek az osztályok vannak implementálva a különböző típusú adatbázisoknak megfelelően, mint például SQL, Oracle, stb (az adatforrástól függően változnak az osztályok nevei, és a névterek, amiben megtaláljuk őket). Ha létrehozunk egy Dataset-et (amit célszerű a fejlesztőkörnyezet Varázslójával megtenni), akkor az biztosítja a megfelelő DataAdaptert, mely tartalmazza az ún. ConnectionString-et (ez tartalmazza az adatbázis eléréséhez szükséges provider nevét,

szervercímet, adatbázisnevet, stb), a CRUD (create, read, update, delete) parancsokat végrehajtó megfelelő Command objektumokat (InsertCommand, SelectCommand, UpdateCommand, DeleteCommand). Ezeket a Command objektumokat köthetjük a szerveren található tárolt eljárásokhoz is, de írhatunk saját query-t is. Ha tehát valamilyen műveletet szeretnénk végezni az adatbázison, akkor a Dataset segítségével lepéldányosítjuk a DataAdaptert (a Dataset-en keresztül érjük el az osztályokat), mely kapcsolatot teremt az adatbázissal, majd meghívhatjuk valamelyik parancsot (ezt elérjük, mint az adapterpéldány egy sima metódusát), mely a meghatározott adatbázisbeli parancsokat hajtja végre, és visszaadja a visszatérési típusának megfelelő adatot. Az előbb említett 4 alapvető adatbázis-műveleten kívül tetszőleges számú tárolt eljárást (vagy megírt lekérdezést) elérhetünk a Dataset, illetve a DataAdapter segítségével (már amennyiben megadtuk őket). A Dataset bevezetésének célja az volt, hogy kapcsolat nélkül tudjunk dolgozni az adatbázisból kinyert adatokon. Ha Dataseten keresztül feltöltünk egy táblát adatokkal, a művelet után a kapcsolat megszűnik az adatbázissal. A memóriában tárolt adatokat módosíthatjuk, törölhetjük, új sorokat szúrhatunk be, de nekünk kell befrissíteni a módosításokat az adatbázisba a megfelelő függvény segítségével. Rengeteg előnye van a Dataset-nek, de ami miatt igazás szeretni kell, az az ún. Típusos Dataset. Az alapvető probléma a “sima”, nem típusos Dataset-tel, hogy ha a lekért adatokkal akarunk dolgozni, akkor a tábla egy adott oszlopára valahogy így kell hivatkoznunk row.Columns[“oszlopnév”]. Ezzel ugye az gond (azon kívül hogy nem is szép), hogy fordítási időben semmi sem derül ki, tehát ha valamilyen hiba lép fel, akkor az futási időben jelentkezik. A Típusos Dataset erre kínál megoldást. A Típusos Dataset a Dataset osztályból származik, így minden funkciójával rendelkezik, és mindenhol használhatjuk, ahol Dataset-et szeretnénk használni. Egy .xsd file-ban tárolt információkat használ a Visual Studio, hogy egy új, erősen típusos Dataset osztályt hozzon létre. A schema (amely az adatbázist írja le) alapján különböző osztályokat és property-ket generál le, melyek megjelennek ebben az új Dataset osztályban. Ezek az osztályok a tábla (amihez kötve vannak) neve alapján generálódnak és mind típusosak, azaz a táblának megfelelő típusos DataTable, DataRow, stb jön létre. Ennek eredményeképp már az adott típusos DataRow osztályban a tábla oszlopai, mint property-k ki lesznek vezetve, így fordítási időben elérjük bármelyik oszlopot a következőképp: row.OszlopNév. Ez csökkenti a hibák lehetőségét, és programozástechnikailag is szebb, átláthatóbb megoldás.

Nézzünk egy példát:

```
public class DataInterface
{
      private static UserTableAdapter pUserAdapter;

      internal static UserTableAdapter UserAdapter
        {
            get
            {
                if (pUserAdapter == null)
                {
                    pUserAdapter = new UserTableAdapter();
                }
                return pUserAdapter;
            }
        }
…
```

A `pUserAdapter` egy privát adattag, melyben a lepéldányosított `UserTableAdapter` objektumot tároljuk. A `UserAdapter` az a property, amin keresztül elérjük az adaptert, ez ellenőrzi, hogy létezik-e már példány belőle, ha nem, akkor lepéldányosítja.

```
…
public static RRFDataSet.UserDataTable SelectUsersByRoleId(int roleId)
{
      RRFDataSet.UserDataTable table = UserAdapter.GetData(null, roleId, null,
      null, null, null);

      return table;
}
```

Ez egy olyan függvény, mely a DataAdapter-en keresztül meghívja a GetData(...) metódust, mely egy select lekérdezést valósít meg, a megfelelő tárolt eljárással. Amikor a DataSet-et generáljuk, akkor a megadott select tárolt eljárás alapján létrehoz alapértelmezett módon egy Fill() és egy GetData() függvényt (ezeket természetesen át lehet nevezni), mindkettő egy adott DataTable-t tölt meg adattal. Az adatbázisban található `uspReadUser` tárolt eljárás paraméterezhető, a User tábla tetszőleges oszlopára, illetve oszlopainak kombinációjára szűrhetünk, így ez a GetData() függvény (és a mögöttes tárolt eljárás) valósítja meg az összes, általunk használt lekérdezést a User táblára vonatkozóan. A GetData() paraméterei vagy adott értéket vesznek fel, vagy `null`-t, ha valamelyik paraméter értéke `null`, akkor arra az oszlopra az összes sort visszaadjuk.

Az előbbi példában csak egy `roleId` paramétert adunk át, így az összes felhasználót visszaadja, aki az adott szerepkörrel rendelkezik. A visszatérési érték egy típusos DataTable objektum. Az üzleti szinten ezt a táblát bejárva hozzuk létre a User objektumok listáját, amivel már tudunk dolgozni magasabb szinteken is. Ha olyan lekérdezést indítunk, ami csak egy sort ad vissza, akkor annak visszatérési értéke egy DataTier-beli User objektum lesz:

```
public static User FindById(int userId)
{
	RRFDataSet.UserDataTable table =
	DataInterface.UserAdapter.GetData(userId, null, null, null, null, null);

	if (table.Rows == null || table.Rows.Count == 0)
	{
		return null;
	}

	RRFDataSet.UserRow row = table.Rows[0] as RRFDataSet.UserRow;

	return new User(row);
}
```

Az új User objektumot úgy állítjuk elő, hogy a paraméterrel rendelkező constructor-t hívjuk meg, mely típusos Datarow-t vár paraméterként. DataTier szinten minden actor objektum tartalmaz egy megfelelő típusos Datarow adattagot, melyen keresztül az összes táblabeli oszlop elérhető, mint property:

Példa: DataTier.Actors.User.cs

```
internal protected RRFDataSet.UserRow row;
public int Id
{
	get
	{
		return row.Id;
	}
	set
	{
		;
	}
}

public int RoleId
{
	get
```

```
        {
            return row.RoleId;
        }
        set
        {
            row.RoleId = value;
        }
}
…
```

A constructor:

```
public User(RRFDataSet.UserRow userRow)
{
      row = userRow;
}
```

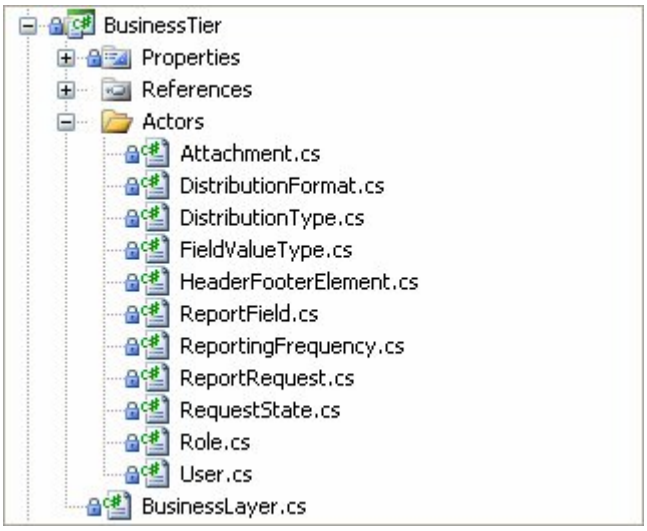
Rendelkezésre állnak még a lekérdező metódusokon kívül az adatmódosító metódusok is, melyeket a típusos TableAdapter nyújt nekünk (a megfelelő tárolt eljárások kötéseivel):

```
public void Update()
{
      DataInterface.UserAdapter.Update(row);
}

public void Insert()
{
      DataInterface.UserAdapter.Insert(row.RoleId, row.UserName,
      row.FirstName, row.LastName, row.EmailAddress);
}

public void Delete()
{
      DataInterface.UserAdapter.Delete(row.Id);
}
```

Láthatjuk, hogy ezen a szinten csak lekérdezzük (illetve módosítjuk) az adatokat, és semmilyen üzleti döntés nem születik. A DataTier minden fentebbi rétegtől el van szeparálva, csak adatokat szolgáltat a BusinessTier számára, mely alkalmazásunkban az üzleti réteget valósítja meg.

### 2.1.3 Business logic layer

Ez a köztes réteg a megjelenítő felület és az adatkezelő réteg között.




**13. ábra: Az üzleti réteg**

Az Actor könyvtárban található osztályok a DataTier-es megfelelőikből származnak, de kiegészítjük olyan plussz információkkal az adott osztályt, melyekre üzleti szinten szükség van. Ezek általában adatbázisban nem tárolt, származtatott, kiszámolt értékek, illetve olyan tulajdonságokat is bevezetünk a leszármazott osztályokban, melyek logikai kapcsolatot hoznak létre egy másik osztállyal. Így egy adott objektumhoz tartozó másik objektum elérhető property-ken keresztül, megkönnyítve ezzel munkánkat, mivel nem kell azon a helyen, ahol felhasználjuk, nekünk lekérdezni az adatbázisból, ezt megteszi helyettünk az adott property `get()` metódusa.

Példa: BusinessTier.Actors.User.cs

```
public Role Role
{
	get
	{
		return BusinessTier.Actors.Role.FindById(this.RoleId);
	}
}

public string RoleName
{
	get
	{
		return Role.Name;
	}
}

public string FullName
{
```

```
        get
        {
                return base.FirstName + " " + base.LastName;
        }
}
…
```

Üzleti szinten is találkozhatunk azokkal a függvényekkel, melyek a megfelelő adatokat szolgáltatják. Minden DataTier-beli metódus ki van vezetve üzleti szintre, vagy a megfelelő Actor osztály vagy a BusinessLayer.cs statikus metódusaként.

Példa: BusinessTier.Actors.User.cs

```
public static new User FindById(int userId)
{
        return CreateInstance(DataTier.Actors.User.FindById(userId));
}
…
```

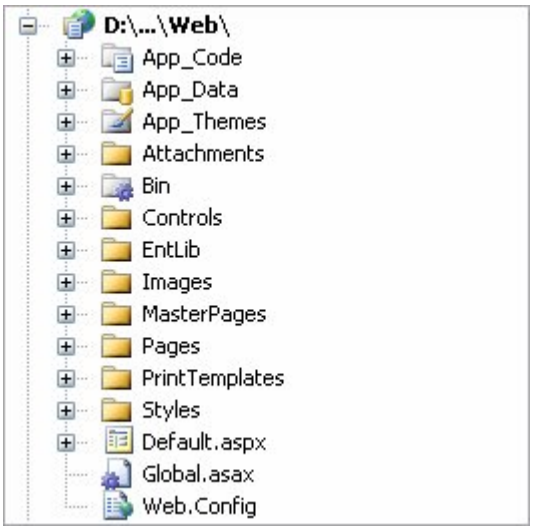
BusinessTier.BusinessLayer.cs

```
public static List<User> SelectUsersByRoleId(int roleId)
{
        List<User> list = new List<User>();
        RRFDataSet.UserDataTable table =
        DataTier.DataInterface.SelectUsersByRoleId(roleId);

        foreach (RRFDataSet.UserRow uRow in table)
        {
                list.Add(User.CreateInstance(new DataTier.Actors.User(uRow)));
        }
        return list;
}
…
```

A User.cs osztálybeli CreateInstance() függvény egy statikus függvény, és a constructor szerepét tölti be. Minden függvényhívás egy DataTier-beli függvényhívásban végződik, így látható, hogyan veszi igénybe az üzleti réteg az alacsonyabb réteg szolgáltatásait. Ezen a szinten teljesen el van fedve az adatszintű megvalósítás, semmit nem kell tudnunk a mögöttes adatbázisról, és az azt érintő lekérdezésekről. Ha más típusú adatbázis szerverre helyezik át az adatbázisunkat, az csak az adatkezelő réteget érinti, ezen a szinten változtatásokra nincs szükség. Az üzleti rétegben csak az alkalmazást érintő szabályoknak megfelelő döntéseket kell meghoznunk.

### 2.1.4 Presentation layer

Az alsó két réteg már képes megfelelően kiszolgálni a megjelenítő réteget, így hát lássuk, hogy épül fel a felhasználói felület. Mint már említettem, ebben az alkalmazásban az adatok megjelenítéséért és a felhasználói interaktivitás kezeléséért ASP.NET lapok felelnek, melyeket az ún. Web Form-okon állíthatjuk össze. A Web Form tetszőleges kontrolokat tartalmazhat, melyek megvalósítják a szükséges felhasználói interfészt és tartalmazzák a szükséges funkciókat. Ezeket a kontrolokat akár tovább is fejleszthetjük saját igényeink szerint, vagy készíthetünk teljesen újakat is. Lássuk hogy épül fel a UI réteg!




**14. ábra: A megjelenítési réteg**

Az App_Code könyvtárba sima osztályokat tartalmazó file-ok kerülnek, melyekre szükségünk lehet az alkalmazásban. Jelen esetben ezek az osztályok különböző hasznos funkciókat valósítanak meg, nem is nagyon részletezem őket, csak említés szintjén:

Authnetication.cs: a felhasználó azonosításában és beléptetésében játszik szerepet.
Email.cs: email küldéssel kapcsolatos metódusokat tartalmaz.
MessageBox.cs: figyelmeztető ablakok megjelenítésére szolgál.
Printer.cs: neve ellenére a ReportRequest-ből előállított Word dokumentum létrehozására szolgál.
SessionHandler.cs: célja a Session kezelés megkönnyítése.

Utils.cs: az alkalmazás szempontjából egyéb, hasznos függvényeket tartalmaz.

Az App_Data könyvtárban egy file található, ez az EMail.config. Mint ahogy neve is mutatja, ez egy konfigurációs állomány, melyben az email küldéssel kapcsolatos beállításokat találjuk.

Az Attachments könyvtár tárolja a szerveren a ReportRequest-hez csatolt állományokat, könyvtárakba rendezve.

A Controls mappa ad helyet az összes saját control-nak, melyek alkalmazás-specifikus feladatokat látnak el.

Az EntLib könyvtár tartalmazza a logfile-okat.

Az Images könyvtárban az alkalmazásban felhasznált képeket tároljuk.

A Pages mappában a weblapokat, vagyis a web form-okat találjuk.

A PrintTemplates tartalmazza azt a template file-t, mely alapján generáljuk a report-hoz kapcsolódó céges dokumentumot, mely tartalmazza a report teljes leírását.

A gyökérkönyvtárban van még a Default.aspx, a Global.asax és a Web.config állomány. A Default.aspx megfeleltethető egy weboldal index.html-jének ASP.NET környezetben, ez az alapértelmezett weblap a webes alkalmazásoknál. A Global.asax-ban alkalmazás szintű objektumokat hozhatunk létre, illetve alkalmazás szintű funkciókat valósíthatunk meg. A Web.config egy XML nyelvre alapozott, konfigurációs beállításokat tartalmazó állomány. Ezen keresztül írhatjuk le programunk alapvető beállításait, vagy akár saját konfigurációs adatokat is megadhatunk.

Mivel az alkalmazás során code-behind modellt használok, ezért egy weblap (illetve control) 2 részből áll, egy .aspx (illetve .ascx) és egy .cs file-ból. Az aspx file többnyire a vizuális elemeket

tartalmazza (esetlegesen valamilyen kliens oldali script-kódot), a cs kódfile pedig az eseménykezelő és egyéb segédmetódusokat.
Vegyünk egy egyszerű példát (nem az alkalmazásból), melyen keresztül könnyebben megérthető az ASP.NET működése.

Példa: Test.aspx
Ha hozzáadunk egy új weblapot a projektünkhöz, akkor automatikusan létrehozza a mögötte álló kódfile-t is, és mindkét állományba kódrészletet generál. Nézzük először a Test.aspx filet!

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Test.aspx.cs"
Inherits="Pages_Test" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    </div>
    </form>
</body>
</html>
```

Ez a fejlesztőkörnyezet által automatikusan generált tartalom.
Az első sorban található `Page` direktíva weblap-specifikus attribútumokat határoz meg, melyeket az ASP.NET parser és fordító használ. Itt találjuk a `CodeFile="Test.aspx.cs"` bejegyzést, mely a mögötte álló kódfile nevét tartalmazza, és a fordító az `Inherits` attribútum segítségével ez alapján tudja, hogy melyik kódfile tartozik a weblaphoz, mely ugye a fordításnál játszik szerepet.
A direktíva alatt látjuk a generált „alap” weblapot, mely egy HTML nyelven megírt oldal. Az egyetlen különbség egy szokványos oldalhoz képest, hogy találunk a form-hoz egy `runat="server"` bejegyzést, mely azt jelöli, hogy ez egy szerver oldali control. Asp.net-es control-ok esetében erre az attribútumra mindig szükség van, különben a fordító hibát jelez, de

HTML control-ok, tag-ek esetén is használhatjuk, ekkor az adott objektum szerver oldalon is elérhető lesz. Ebben a forrásfile-ban csak HTML elemeket találunk, de tetszőlegesen vegyíthetjük asp-s control-okkal is: tegyünk bele egy asp-s linkbutton-t!

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Test.aspx.cs"
Inherits="Pages_Test" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:LinkButton ID="LinkButton1" runat="server"
OnClick="LinkButton1_Click">LinkButton</asp:LinkButton>
    </div>
    </form>
</body>
</html>
```

Kliens oldalon a következő forráskód generálódik:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head><title>
      Untitled Page
</title></head>
<body>
    <form name="form1" method="post" action="Test.aspx" id="form1">
<div>
<input type="hidden" name="__EVENTTARGET" id="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" id="__EVENTARGUMENT" value="" />
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwULLTE4NDIzMDg2NDRkZGKCbxsB0VJTvAn0jGd3ecJJJ4fx" />
</div>

<script type="text/javascript">
//<![CDATA[
var theForm = document.forms['form1'];
if (!theForm) {
    theForm = document.form1;
}
function __doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
```

```
        theForm.__EVENTTARGET.value = eventTarget;
        theForm.__EVENTARGUMENT.value = eventArgument;
        theForm.submit();
    }
}
//]]>
</script>


    <div>
      <a id="LinkButton1"
href="javascript:__doPostBack('LinkButton1','')">LinkButton</a>
    </div>

<div>

      <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
value="/wEWAgL20vG4BQLM9PumD2YQaaAX+JXRAWhZC5n6BtVXs3pc" />
</div></form>
</body>
</html>
```

Megfigyelhető, hogyan „cserélődtek le” az asp-s elemek HTML elemekre. A form megkapta a `method` és `action` attribútumokat. A .NET-es LinkButton-ból HTML-es <a> tag lett, ugyanis minden ASP.NET-en belüli control-nak van HTML-en belüli megfelelője, illetve minden szerver oldali control helyett HTML kódot állít elő a szerver. Egyszerűbb control-ok esetében egy az egyben történik a megfeleltetés valamilyen HTML taggel, de vannak olyan control-ok is, amelyek összetettebb HTLM kódot eredményeznek.

Bekerült még négy hidden field, mely adattárolásra szolgál. Az `__EVENTVALIDATION` hidden field most számunkra nem lényeges, az igazán fontos és érdekes az `__EVENTTARGET`, az `__EVENTARGUMENT` és a `__VIEWSTATE`. A `__VIEWSTATE` szolgál arra, hogy webes alkalmazásunkban a különböző kérések között a control-ok aktuális állapota megmaradjon, ugyanis mint az ismeretes, két kérés egymáshoz képest független a webes világban, így a különböző elemek állapota is elveszik, ez adja az internet állapotmentes jellegét. Ezt küszöböli ki az ASP.NET ezzel a rejtett mezővel, mely utazik a kérések között, és mielőtt visszaküldi a kliensnek a választ a szerver, beállítja a control-t a `__VIEWSTATE` objektumban tárolt értékeknek megfelelően. A másik kettő hidden field a javascript kódrészletben kap szerepet, mely az eseménykezeléshez szükséges. Ha egy olyan control-t teszünk ki az oldalra, melynek használjuk valamelyik eseményét, akkor a szerver a következő script-et generálja ki a kliensnek:

```
<script type="text/javascript">
//<![CDATA[
var theForm = document.forms['form1'];
if (!theForm) {
    theForm = document.form1;
}
function __doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
        theForm.__EVENTTARGET.value = eventTarget;
        theForm.__EVENTARGUMENT.value = eventArgument;
        theForm.submit();
    }
}
//]]>
</script>
```

Ha megnézzük a `__doPostBack(eventTarget, eventArgument)` függvényt, láthatjuk, hogy két paramétert vár. A paraméterek értékét beállítja a megfelelő rejtett mezőkbe, mely postback esetén visszakerül a szerverhez. Ez alapján tudja, hogy melyik control melyik eseménye váltotta ki a postback-et (ezekre vonatkozó információt tárol el bennük). Ez a javascript függvény pedig az adott esemény eseménykezelője lesz kliens oldalon a megfelelő controlnál. Az `eventTarget` értéke mondja meg, hogy melyik control váltotta ki a submit-et, az `eventArgument` pedig az eseménnyel kapcsolatos további adatot tárol.

Példánk esetében szerver oldalon a LinkButton OnClick eseményére iratkoztunk fel, az eseménykezelő metódus a Test.aspx.cs file-ban található. Kliens oldalon az ASP.NET beköti nekünk ezt a javascript függvényt a link `href` attribútumára a megfelelő paraméterekkel, így ha rákattintunk az adott linkre, az visszaküldi az oldalt a webszervernek, a szerver a rejtett mezők értéke alapján pedig tudja, melyik esemény váltódott ki és melyik eseménykezelő(ke)t kell futtatnia.

Amikor egy kérdés beérkezik a szerverhez, az oldal különböző állapotokon megy át, ezt nevezzük életciklusnak. Az életciklus során különböző események váltódnak ki, melyeknél érvényes az automatikus esemény-feliratkozás (ezt állítjuk az `AutoEventWireup` attribútummal), azaz hogy az ASP.NET az események kiváltódásakor meghatározott nevű eseménykezelőket keres és futtat, tehát nem kell nekünk explicit feliratkoznunk az eseményekre. Ebben az esetben az eseménykezelő függvények neve `Page_esemény` alakú, mint például `Page_Init` vagy `Page_Load`.

Összességében elmondható, az ASP.NET és a framework rengeteg terhet levesz a fejlesztő válláról, és az előbb említett hasonló kényelmes megoldásokkal segíti a programozási munkát.

## 2.2 Az alkalmazás funkcionalitása

A ReportRequest alkalmazás tehát egy szabványos kommunikációs csatorna az alkalmazottak számára, melyen keresztül report-kéréseket küldhetnek a Reporting Team-nek. Az alkalmazás lehetővé teszi a kérést indító felhasználó számára, hogy biztosítsa az összes információt, melyre a Reporting Team-nek szüksége lehet a report elkészítéséhez.

### 2.2.1 Felhasználói szerepkörök

Az alkalmazás 3 szerepkört támogat:

- Requestor: A Verety vállalat bármelyik alkalmazottja lehet, akinek szüksége van report-ra. A Requestor a ReportRequest alkalmazáson keresztül küldi el report-igényeit.
- Administrator / Business Analyst: Administrator szerepkörrel rendelkezik a Reporting Team development manager-e vagy business analyst-ja, kinek legfontosabb feladata az igényelt report-ok megvalósíthatóságának kiértékelése. Ezen felül az Administratornak teljes joga van a report kérések felett.
- Developer: A Developer olvasási joggal rendelkezik a rendszerben, ő készíti el a report-okat a kérések alapján.

### 2.2.2 Report állapotok

Az alkalmazásban a kérések az elkészítés időpontjáig különböző állapotba kerülhetnek:

- Opened: új report alapértelmezett állapota.
- Submitted: elküldött report állapota.
- Declined: elutasított report állapota.
- Approved/Prioritized: elfogadott report állapota.

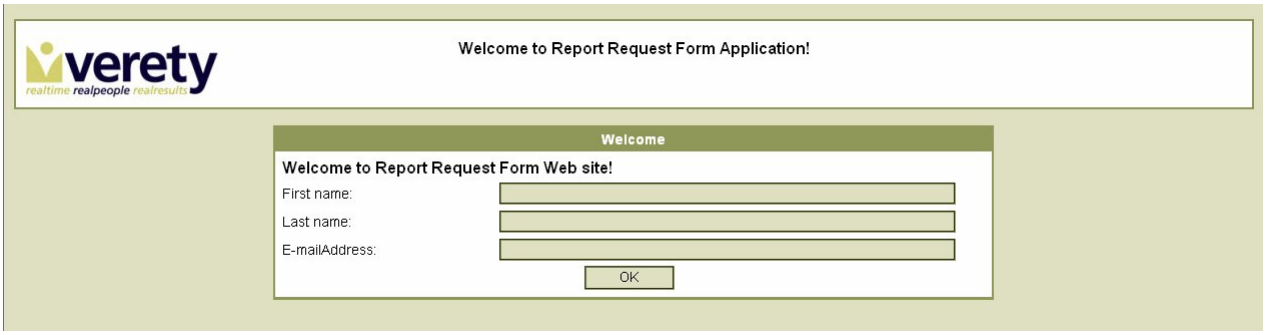
- In-progress: elkészítés alatti report állapota.
- Ready: elkészült report állapota.
- Closed: lezárt report állapota.

Az állapotváltozás menetét a függelékben található kép mutatja (Függelék 1.kép).

### 2.2.3 Fontosabb felhasználói képernyők

Welcome képernyő:

A rendszer windows authentication-t használ, így a kliens windows-os felhasználói fiókját használja a beléptetéshez. Ha a felhasználó még nem regisztrált felhasználó, akkor a Welcome képernyőn alapvető adatait biztosítva regisztrálhat. Ha a felhasználó regisztrálta magát korábban, akkor a rendszer automatikusan belépteti és a szerepköréhez tartozó főoldalra navigálja.



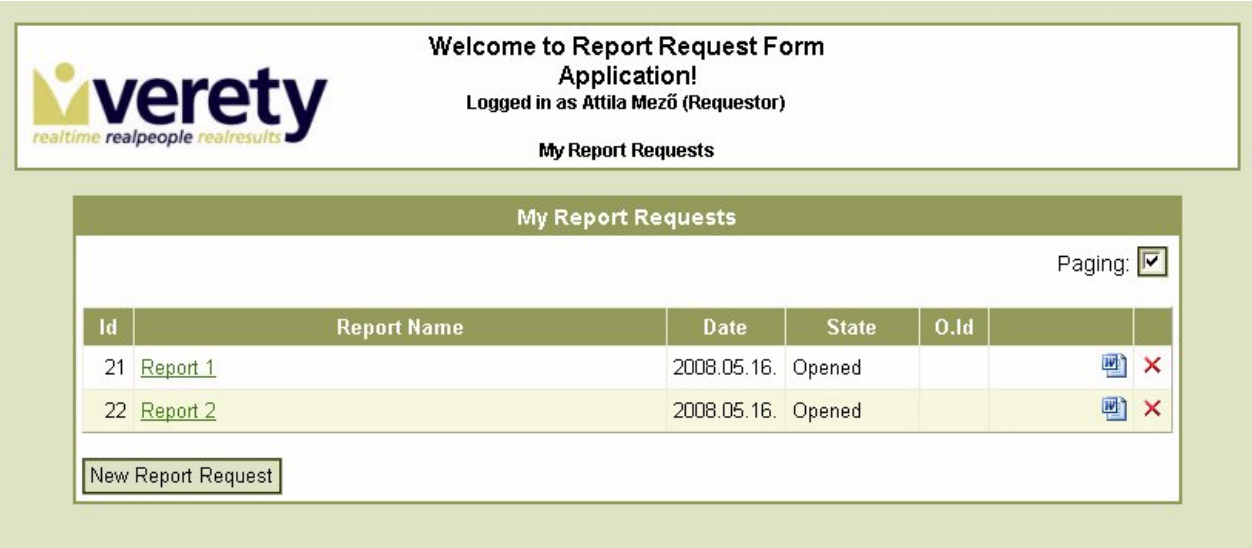

**15. ábra: Welcome képernyő**

Szerepkörhöz tartozó főképernyők:

Belépés után a felhasználók a megfelelő nyitólapra kerülnek.

Requestor:

Ez a képernyő mutatja a ’My Report Requests” listát, mely a felhasználóhoz tartozó elküldött, megnyitott, elutasított és lezárt report kéréseket tartalmazza.




**16. ábra: Requestor home**

A lista a következő oszlopokat tartalmazza:

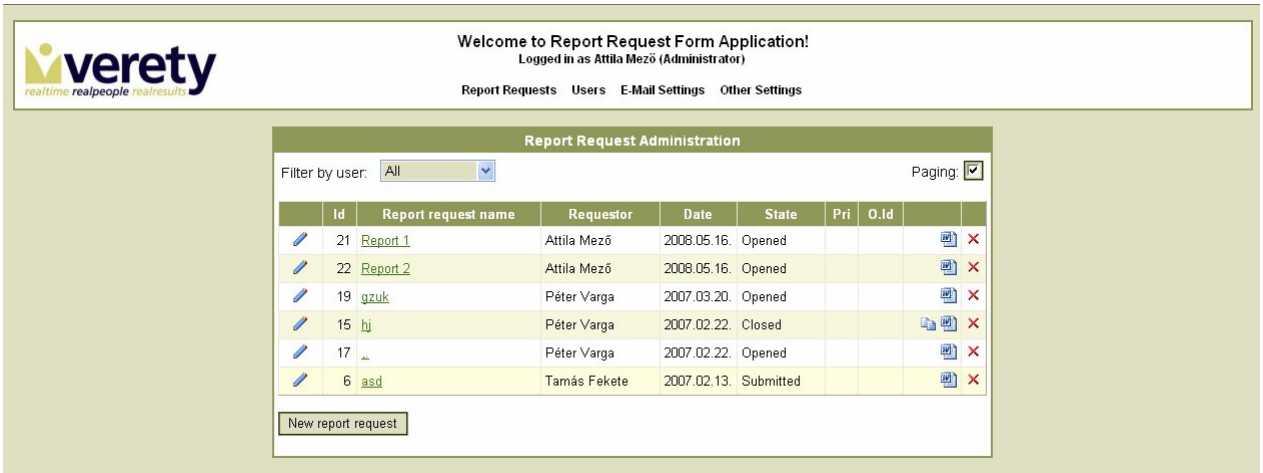
- Id: a rendszer által generált Id.
- Report Name: a report neve.
- Date: a létrehozás dátuma.
- State: a report aktuális állapota. Lehetséges értékek: Open, Submitted, Declined, Approved, In progress, Ready, Closed.
- O.Id: másolat készítése esetén az eredeti ReportRequest-hez tartozó Id.
- Műveleteket tartalmazó oszlop: például nyomtatás.
- Törlés

Ha valamelyik report-ra klikkel a felhasználó, akkor a report request form jelenik meg VIEW üzemmódban, ekkor csak olvasásra nyitja meg a report-ot. Szerkeszteni csak akkor tudja, ha a report állapota Opened, Declined, Ready vagy Closed, ha Submitted, In Progress vagy Approved akkor erre nincs lehetőség.

Ezen az oldalon tud a felhasználó létrehozni új report request-et.

Administrator:

Ez a képernyő mutatja a ’All Report Requests” listát, mely az összes elküldött, megnyitott, elutasított és lezárt report kérést tartalmazza.




**17. ábra: Administrator home**

A lista ugyanazokat az oszlopokat tartalmazza mint a ’My Report Requests’ lista, kibővítve újabb oszlopokkal:

- Szerkesztés
- Requestor: a felhasználó neve, aki a report-ot kérte.
- Pri: a report prioritása
- Művelet oszlop: újabb művelet a másolat készítése adott report-ról.

Ha a lista valamelyik elemére klikkel a felhasználó, akkor a report request form jelenik meg VIEW üzemmódban.

Developer:

Ez a képernyő mutatja a 'Requests Assigned to Me' listát, mely az összes olyan nyitott report kérést tartalmazza, amit egy Administrator felhasználó rendelt a fejlesztőhöz.


**18. ábra: Developer home**

A lista hasonló oszlopokat tartalmaz mint a 'My Report Requests' lista, de nincs törlési lehetőség. Új oszlop:

- Pri: a report prioritása

RRF Wizard

A Report Request Creation Wizard segítségével hozhatunk létre új report request-et. A wizard négy oldalból áll, melyeken a report-hoz szükséges adatokat biztosítjuk. A négy oldal a következő:

- Report Summary
- Report Headers / Footers
- Report Field Specification
- Attachments

A wizard navigálásához vertikális menü áll rendelkezésünkre, mely négy linket tartalmaz a négy oldalra. A megfelelő linkre kattintva a felhasználó bármelyik oldalra léphet. Az oldalakon található gombok a report állapotától és a felhasználó szerepkörétől függően jelennek meg.

A következő gombok jelenhetnek meg:

- Edit: Requestor szerkeszthet vele OPENED vagy DECLINED állapotú, Administrator pedig RE-OPENED állapotú report kéréseket. Ez esetben a Wizard EDIT módba vált.
- Submit: Requestor küldhet el OPENED állapotú report kérést a Reporting Team-nek, vagy Developer jelezheti egy IN-PROGRESS állapotú report elkészültét.

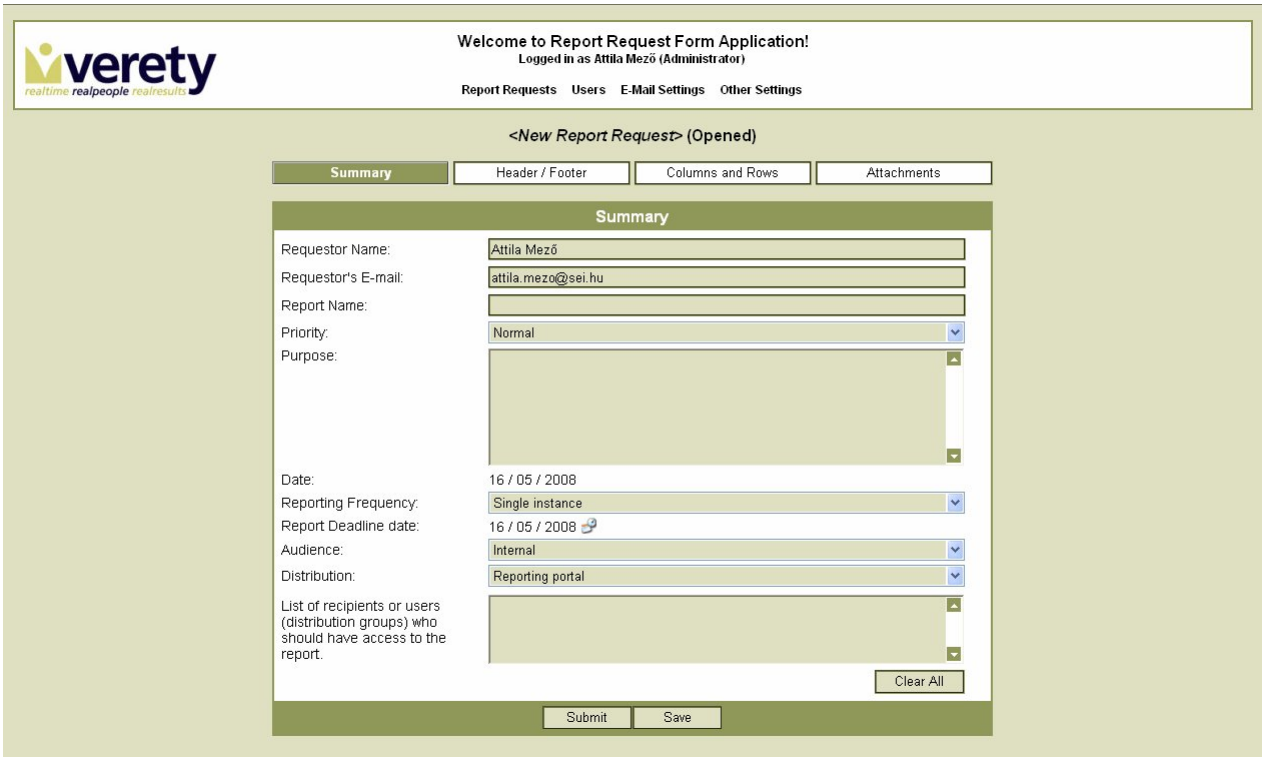
- Approve: Administrator jóváhagyhat egy SUBMITTED állapotú report kérést, mely során megkezdődik a report elkészítése.
- Decline: Administrator visszautasíthat egy SUBMITTED állapotú report kérést.
- Re-open: Administrator újra megnyithat egy CLOSED állapotú report kérést.

RRF Wizard Page 1 (Report Summary)

Az első oldalon a report általános adatait adhatjuk meg.

- Requestor Name: A Requestor neve.
- Requestor's E-mail: A Requestor email címe. Alapértelmezett érték a @verety.com, ezt a felhasználó módosíthatja.
- Report Name: A report neve.
- Priority: A report prioritása. Legördülő lista a következő értékekkel: <Immediate | High | Normal | Low>.
- Purpose: A report célja.
- Date: Csak olvasható mező, az aktuális dátumot mutatja. Formátum: <DD / MM / YYYY>.
- Reporting Frequency: A reportot milyen időközönként kell előállítani. Legördülő lista a következő értékekkel: <Single Instance | Daily | Weekly | Monthly> (illetve további 'Other' érték a kivételes esetek kezelésére: pl. kéthetente).
- Report Deadline date: A report befejezésének határideje. Nincs alapértelmezett érték, formátum: <DD / MM / YYYY>.
- Audience: A report belső vagy külső használatra készül. Legördülő lista a következő értékekkel: <Internal | External>.
- Distribution: A report publikálásának módja. Legördülő lista a következő értékekkel: <Automatically emailed | Reporting Portal>.
- Format: A report emailben történő küldése esetén a csatolt állomány formátuma, mely a report-ot tartalmazza. Ez a legördülő lista rejtve marad, egészen addig, amíg a Distribution listában ki nem választják az 'Automatically emailed' opciót. Lehetséges értékek: <Adobe PDF | Excel .xls | CSV >.

- List of recipients or users (distribution groups) who should have access to the report: Azon felhasználók listája, akik hozzáférhetnek a report-hoz.

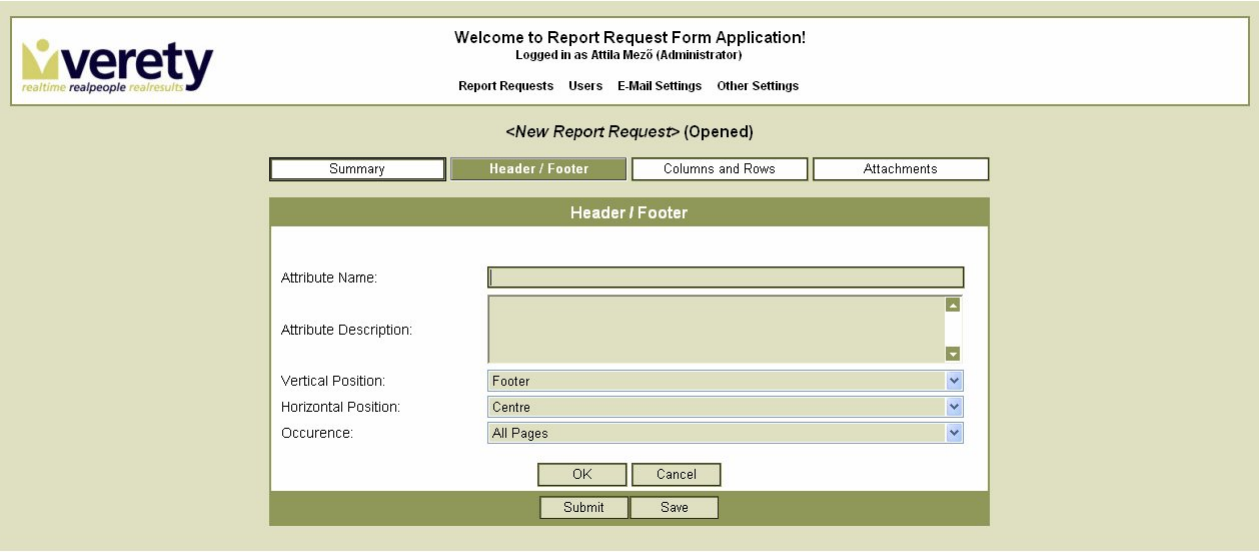



**19. ábra: Wizard Page 1**

RRF Wizard Page 2 (Report Header / Footer)

A második oldalon a report fejlécében, illetve lábjegyzetében megjelenő elemeket definiálhatjuk (pl.: dátum, oldalszám, céges logo, stb). Ehhez a következő beviteli mezők állnak rendelkezésünkre:

- Attribute name: A megjelenítendő elem neve.
- Attribute description: Az elem leírása.
- Vertical position: Legördülő lista a következő értékekkel: <Header / Footer>
- Horiontal position: Legördülő lista a következő értékekkel: <Left / Centre / Right>
- Occurence: Legördülő lista a következő értékekkel: <Front Page / All Pages / Last Page>

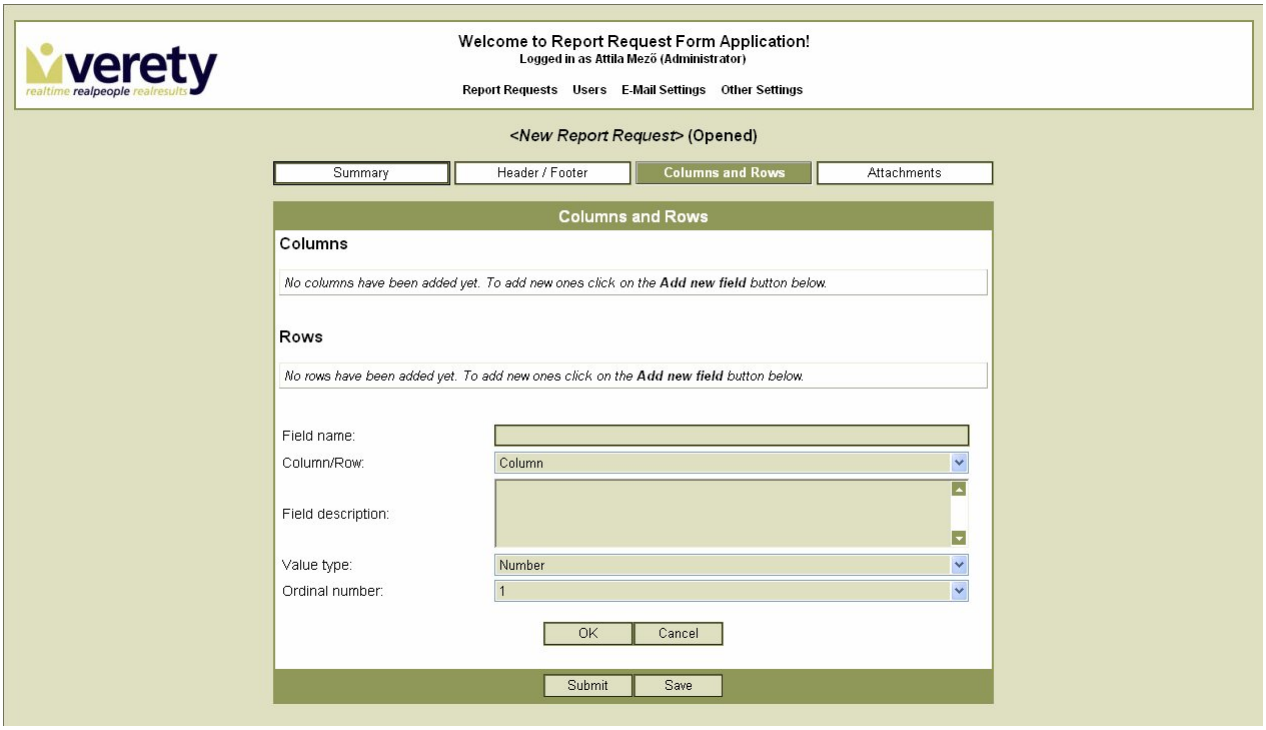



**20. ábra: Wizard Page 2**

RRF Wizard Page 3 (Report Columns and Rows Specification)

Ezen az oldalon adja meg a felhasználó azokat az oszlopokat vagy sorokat és azok leírását, melyek a report-ban megjelennek. Minden ilyen mező 5 tulajdonsággal van leírva:

- Field name: A mező neve.
- Column / Row: legördülő lista, melyben kiválaszthatja a felhasználó, hogy a definiált mező oszlop vagy sor.
- Field description: A mező leírása, megmondja, hogy hogy kell kiszámítani az értékét.
- Value type: legördülő lista a következő értékekkel: <Number, Number%, Text>. Ez határozza meg, hogy az adott mező értéke milyen típusú.
- Ordinal Number: az adott oszlop vagy sor hanyadik oszlop vagy sor legyen.

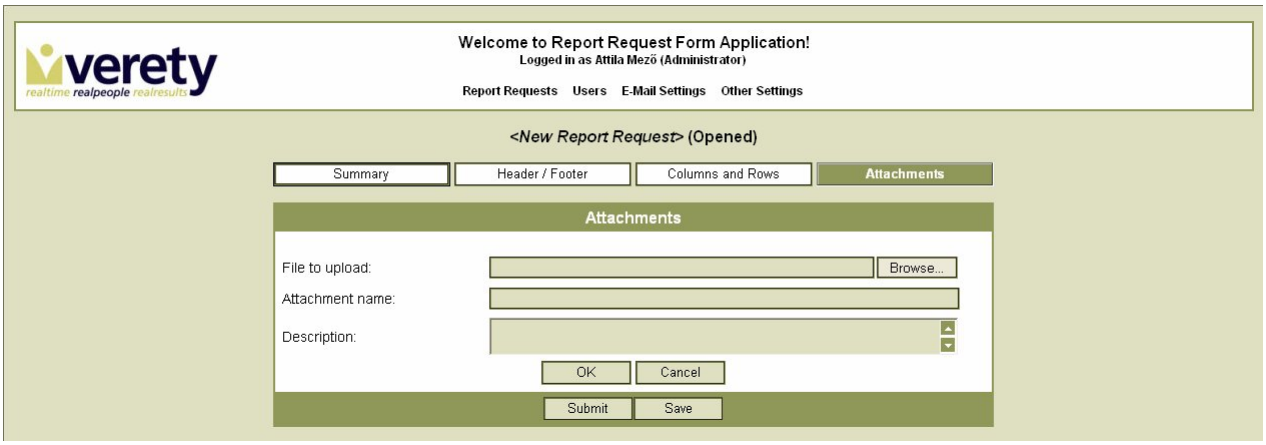



**21. ábra: Wizard Page 3**

RRF Wizard Page 4 (Attachments)

Lehetőség van file-okat csatolni a report kéréshez, mely tartalmazhat bármilyen olyan információt, ami a report elkészítéséhez szükséges (pl. egy vázlatos kép a kért report-ról).

- File to upload: a file amit szeretnénk csatolni.
- Attachment name: a csatolt állomány neve.
- Description: a file leírása.

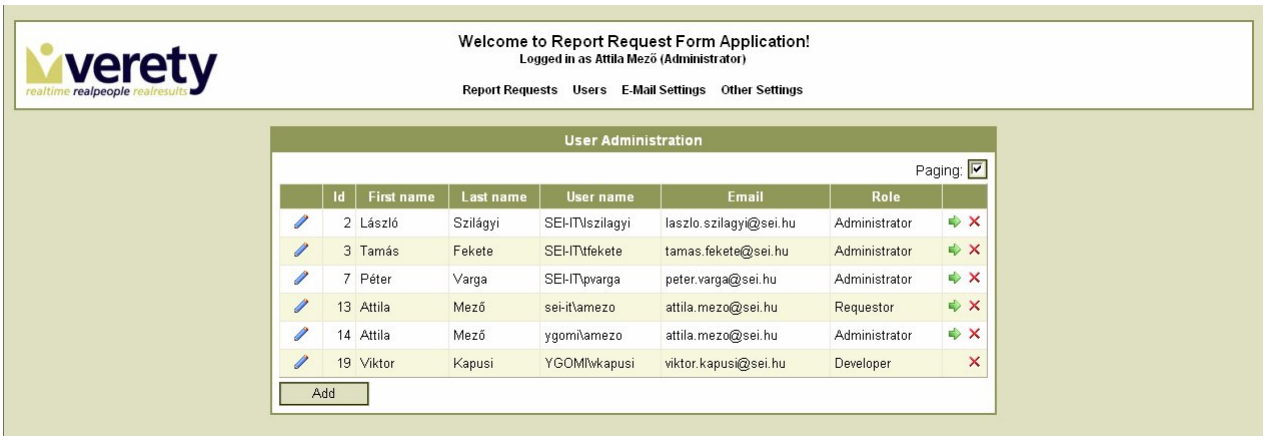



**22. ábra: Wizard Page 4**

Egyéb képernyők:

User Administration

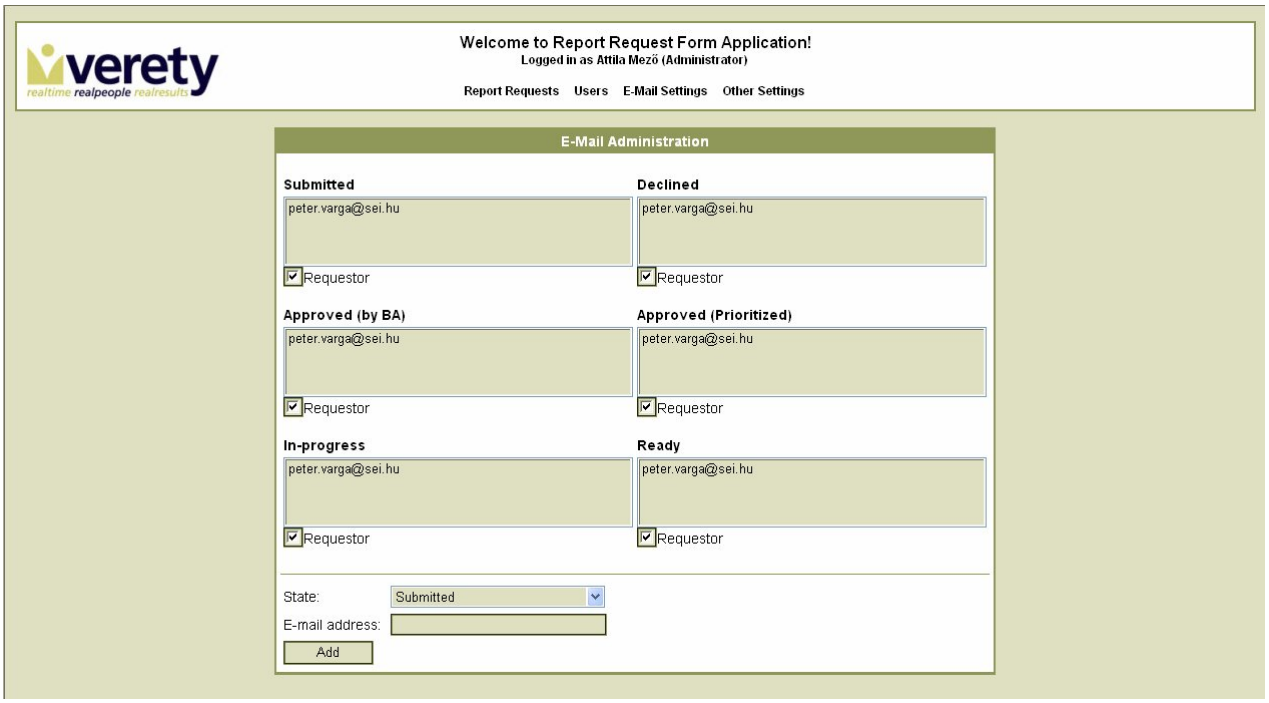
A rendszerben található felhasználók kezelésére szolgál: szerkeszthetünk, törölhetünk, új felhasználót adhatunk az alkalmazáshoz. Ezen keresztül megtekinthetjük a felhasználóhoz tartozó report request-eket is.




**23. ábra: User Administration**

Email Administration

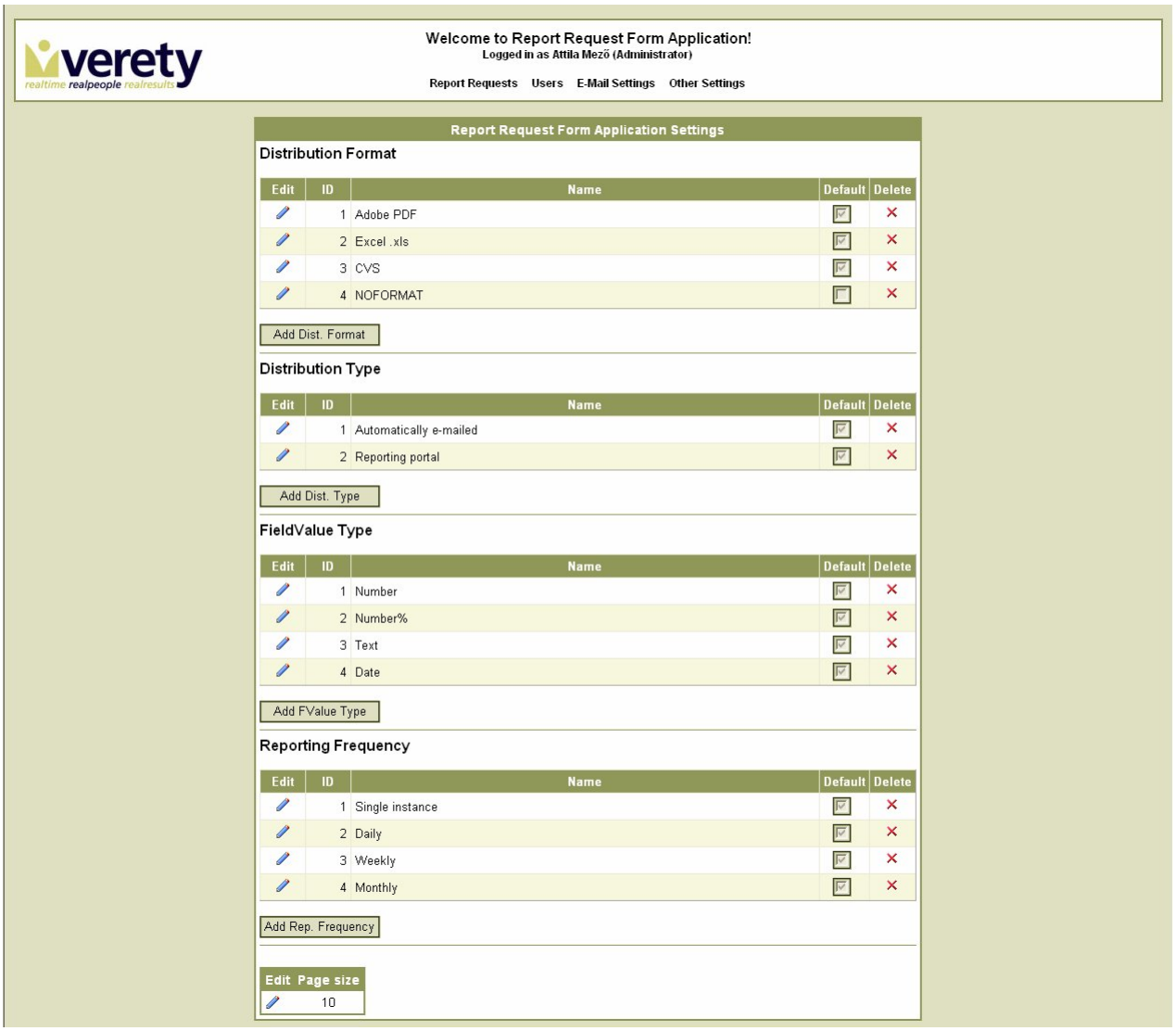
Az email beállításokat szerkeszthetjük. Az alkalmazásban beállíthatjuk, hogy adott állapotváltásról kik kapjanak email-ben értesítést.




**24. ábra: Email Administration**

Report Request Form Application Settings

Egyéb alkalmazásszintű beállításokat módosíthatunk.




**25. ábra: Application Settings**

# ÖSSZEFOGLALÁS

Úgy érzem a követelményeknek sikerült eleget tenni, és az alkalmazás képes betölteni a neki szánt szerepet és közös kommunikációs felületet nyújthat a Reporting Team és a report-okat igénylő alkalmazottak között. A kimutatás állapota folyamatosan végigkísérhető, de az automatikus email értesítések is megkönnyítik a felhasználók dolgát. Az alkalmazás felhasználói felülete letisztult, áttekinthető, így különösebb ismeret nélkül is használható. Hátránya, hogy a teljes funkcionalitás eléréséhez a kliens gépén Microsoft Office telepítése szükséges.

A jövőben az alkalmazás továbbfejlesztésére ad lehetőséget a manapság elterjedt új technológia, az AJAX megjelenése. Segítségével az alkalmazás még inkább felhasználóbaráttá tehető.

# IRODALOMJEGYZÉK

[1] Hatvany Béla Csaba – ASP.NET vezérlők programozása

[2] David. S. Platt – Bemutatkozik a Microsoft .Net

[3] Marco Bellinaso – ASP.NET 2.0 Website Programming: Problem – Design – Solution (Programmer To Programmer) (2006)

[4] Joe Duffy – Professional .NET Framework 2.0 (Programmer To Programmer) (2006)

[5] Ken Schwaber – Agile Project Management With Scrum (Microsoft Professional) (2004)

[6] MSDN Library - .Net Framework
http://msdn.microsoft.com/en-us/library/w0x726c2.aspx

[7] MSDN Library – Common Language Runtime (CLR)
http://msdn.microsoft.com/en-us/library/ms131047.aspx

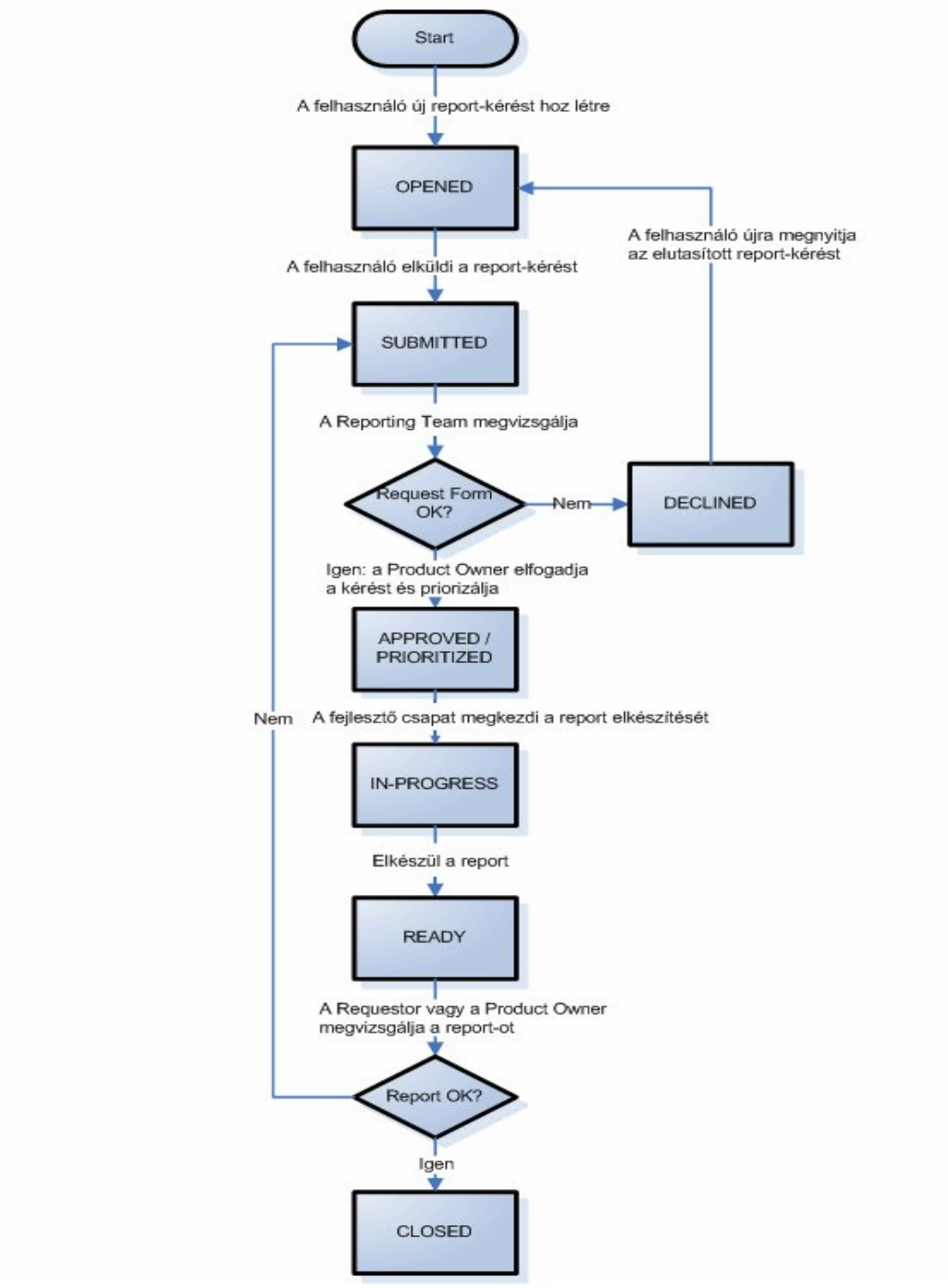
[8] Active Server Pages (ASP)
http://en.wikipedia.org/wiki/Active_Server_Pages

[9] MSDN Library – ASP.NET
http://msdn.microsoft.com/en-us/library/4w3ex9c2.aspx

[10] MSDN Library – Multi-tier Application
http://msdn.microsoft.com/en-us/library/ms978610.aspx

# FÜGGELÉK

1.kép: A report-ok állapotváltozásának folyamatábrája.



2. kép: Adatbázis-diagramm